



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

1971-06

XPL CGP: an XPL-based semantic language processor

Finne, Peter Charles

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/27338>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

XPL CGP: AN XPL-BASED SEMANTIC LANGUAGE PROCESSOR

PETER CHARLES FINNE

LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

XPL CGP: An XPL-based Semantic Language Processor

by

Peter Charles Finne
Lieutenant, United States Navy
B.A., University of Kansas, 1965

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
June 1971

ABSTRACT

The XPL CGP is a complete compiler generator package based on the XPL system. With the introduction of a semantic meta-language (SML) and an associated processor, the package is capable of generating a production compiler for any computer programming language with a mixed strategy precedence grammar. The only input required in most cases is the syntax of the language encoded in BNF and the corresponding semantics encoded in SML. The resulting compiler will generate code which may be executed on a simulated stack-oriented machine.

TABLE OF CONTENTS

I.	INTRODUCTION AND BACKGROUND -----	7
A.	THE PROBLEM -----	7
B.	REVIEW OF CURRENT SYSTEMS -----	8
1.	META-PI -----	8
2.	CC -----	9
3.	FSL -----	10
4.	XPL -----	10
C.	DEVELOPMENT -----	11
II.	THE XPL SYSTEM -----	16
A.	AN OVERVIEW -----	16
B.	THE XPL PARSING ALGORITHM -----	16
C.	THE COMPONENTS OF XPL -----	19
1.	XPLSM -----	19
2.	XCOM -----	21
3.	ANALYZER -----	23
4.	SKELETON -----	25
D.	THE LANGUAGE XPL -----	25
1.	MACRO DEFINITION -----	27
2.	DO CASE CONSTRUCT -----	27
3.	IMPLICIT PROCEDURES -----	27
III.	THE SEMANTIC META-LANGUAGE -----	29
A.	AN INTRODUCTION -----	29
B.	THE CONSTRUCTS OF SML -----	29
1.	SML PROGRAM STRUCTURE -----	29

2.	DECLARATIONS -----	30
a.	TABLES -----	31
b.	STACKS -----	33
c.	CELLS -----	33
d.	TAGS -----	33
e.	STRINGS -----	34
f.	FLAGS -----	34
3.	SML VARIABLES -----	35
a.	BOOLEAN VARIABLES -----	35
b.	ARITHMETIC VARIABLES -----	36
4.	SML OPERATIONS -----	37
a.	ASSIGNMENT STATEMENT -----	38
b.	CELL OPERATIONS -----	39
c.	TABLE OPERATIONS -----	39
d.	STACK OPERATIONS -----	42
e.	CODE OPERATIONS -----	44
f.	STATEMENT CONTROL -----	46
g.	MISCELLANEOUS STATEMENTS -----	47
IV.	THE XPL CGP -----	49
A.	AN OVERVIEW -----	49
B.	SYSTEM INPUT -----	49
1.	CONTROL CARDS -----	49
2.	BNF INPUT -----	51
3.	SML INPUT -----	51
4.	END-OF-FILE -----	51
C.	PRESCANNER -----	52
D.	SYNTACTIC PREPROCESSOR -----	52

E.	SEMANTIC PREPROCESSOR -----	52
F.	SUPERSKELETON -----	53
G.	THE BALGOL-2 MACHINE -----	55
V.	CONCLUSIONS -----	56
A.	THE CAPABILITY OF XPL CGP -----	56
B.	SUGGESTIONS FOR FURTHER RESEARCH -----	57
1.	THE BNF FOR SML -----	57
2.	RESERVED WORD PROBLEM -----	57
3.	INDEFINITE TABLES AND STACKS -----	57
4.	EXTENSIONS TO SML -----	58
a.	AUTOMATIC BLOCKING -----	58
b.	PARAMETERS -----	58
5.	EXTENSIONS TO XPL CGP -----	58
C.	CONCLUDING REMARKS -----	59
APPENDIX A	SKELETON -----	60
APPENDIX B	BNF FOR THE XPL LANGUAGE -----	78
APPENDIX C	BNF FOR SML -----	81
APPENDIX D	THE CGP PRESCANNER -----	85
APPENDIX E	THE SYNTACTIC PREPROCESSOR -----	92
APPENDIX F	THE SEMANTIC PREPROCESSOR -----	129
APPENDIX G	SUPERSKELETON -----	209
APPENDIX H	THE BALGOL-2 INTERPRETER -----	240
APPENDIX I	BNF FOR THE SAMPLE LANGUAGE -----	257
APPENDIX J	CGP INPUT FOR THE SAMPLE LANGUAGE -----	259
APPENDIX K	CGP OUTPUT FOR THE SAMPLE LANGUAGE -----	263
	LIST OF REFERENCES -----	269
	INITIAL DISTRIBUTION LIST -----	270
	FORM DD 1473 -----	271

LIST OF DRAWINGS

1.	CGP I -----	13
2.	CGP II -----	14
3.	CGP III -----	15
4.	XPL SYSTEM AND ENVIRONMENT -----	17
5.	THE XPL ENVIRONMENT FROM THE USER'S VIEWPOINT -----	18
6.	THE XPL COMPILER-GENERATOR SYSTEM -----	20
7.	PROCEDURES OF XCOM -----	22
8.	PROCEDURES OF ANALYZER -----	24
9.	PROCEDURES OF SKELETON -----	26
10.	THE USER'S VIEW OF TABLES AND STACKS -----	32
11.	THE XPL CGP -----	50
12.	THE PRODUCTION OF SEMANTIC PREPROCESSOR -----	54

I. INTRODUCTION AND BACKGROUND

A. THE PROBLEM

In view of the fact that computer-programming in high-level languages became commonplace less than twenty years ago, it is not surprising that opinions should differ on the form and function of these languages. There are some who believe that the language should be made to fit the user's needs and desires. There are others who advocate a single universal language to facilitate communication and efficiency. While both opinions have merit, the current plethora of high-level programming languages does give evidence to the need for a wide variety of programming constructs and data structures that are both adaptable and efficient.

The greatest obstruction to the introduction of such user-oriented languages is, of course, the enormity of the compiler-writing task. Until recently, each compiler had to be laboriously hand-produced in assembly language. However, the advent of the compiler-compiler promises to significantly reduce the time and effort required to place a new language into production.¹

The compiler-compiler is neither simple in concept nor easily implemented. It requires the development and application of some very efficient data and meta-language structures. Since there is little formalism on which to base these structures, the problem of constructing an effective compiler-compiler is doubly difficult.

1. As used herein, the term compiler-compiler is intended to represent only those programs which, given only some form of meta-language input, have the capability to generate a production compiler for any given language on any given machine configuration.

This research effort is envisioned as an investigation of the feasibility of an XPL-based compiler-compiler. The primary effort will be directed at automating the semantic processing phase of the XPL system.¹ A semantic meta-language (SML) will be introduced and a preprocessor will be developed to interpret SML. An analysis of the final product will be made to determine the feasibility of fully automating the system.

B. REVIEW OF CURRENT SYSTEMS

Attempts to develop a compiler-compiler have been based on a wide variety of techniques with varying degrees of success. A survey of some of the more significant efforts will provide some insight into the state of the art.

1. Meta-Pi

The Meta-Pi system [Ref. 1], developed under the direction of J. T. O'Neil, is classified as a syntax-directed symbol processor [Ref. 2]. Meta-Pi is essentially an interactive compiler-generator package which employs a top-down, left-to-right, deterministic parsing algorithm.² Input to the system is in a BNF-like meta-language which allows both syntactic specification and semantic interpretation of the user language. The input is essentially transformed into a sequence of calls to recursive functions which perform syntax checking and code production. The Meta-Pi system appears to suffer from a number of drawbacks, two of which seem

1. It is generally agreed that there are three distinct processes which must be specified in a compiler: the parsing algorithm or syntax checker (syntactics), the semantic processing phase (semantics), and the specification of the language environment (mechanics).

2. The term compiler-generator package (CGP) will be used to specify any compiler-compiler that is limited to a specific class of languages on a specific machine.

significant. The most apparent drawback is the Meta-Pi language; for although the syntactic input elements are straightforward, the variety and uniqueness of the semantic commands would make utilization cumbersome at best. A more significant than apparent drawback is the nature of the parsing algorithm. While Meta-Pi is itself deterministic and hence easily lends itself to top-down, left-to-right analysis, it is probable that any extensive user language is non-deterministic. The implementation of such a language would necessitate time-consuming backup and error-recovery procedures, and invite inefficiency.

2.. CC (Compiler-Compiler)

The CC (Compiler-Compiler) was developed by Brooker and Morris and is one of the oldest compiler-generator packages [Refs. 2 and 3] . The result of the top-down, left-to-right syntax analysis performed by CC on the BNF - like meta-language input is a complete syntax tree. This syntax tree is then collapsed by the application of the semantic routines associated with the syntactic elements. One of the most significant features about CC is its modularity. Thus, while a number of the semantic routines might have to be developed by the user to implement a particular language construct, they can often be simply constructed from other more primitive routines. Additionally, the modularity would appear to allow a relatively straightforward approach to implementing the system on a different machine configuration. The most significant shortcoming of CC is the extremely inefficient analysis algorithm employed. It is intrinsically time and space consuming for most user languages to be implemented.

3. FSL (Formal Semantic Language)

FSL (Formal Semantic Language) was developed by Feldman and represents one of the best-conceived efforts to produce a CGP [Ref. 4] . It is a relatively successful attempt at totally automating the semantic processing phase. One of the most important differences between FSL and its predecessors is the conceptual and physical separation of syntax and semantics of a user language. The input of syntactic specification to FSL is in PL, essentially a superset of BNF. Associated with each syntactic production is a semantic production which is given in the FSL meta-language. As each syntactic element is recognized by the bottom-up analyzer, the associated semantic production is executed. While there are a number of minor shortcomings present in the FSL system, primarily in the absence of some constructs in the meta-language, the motivation of the system is conceptually simple and easily understood. Furthermore, the FSL meta-language is natural, readable, and easily employed.

4. XPL

The XPL compiler-generator system was only recently developed by McKeeman et al. [Ref. 5] and the concepts surrounding its development are quite similar to those of FSL. Like FSL, it is based on a bottom-up parsing algorithm; and like FSL, a semantic interpretation is associated with each syntactic element. However, very much unlike FSL, there is no semantic meta-language inherent in the system to simplify the semantic interpretation (except for the XPL source language). Instead, the user is required to completely specify nearly all compile and run time activities not associated with the syntactic analysis. This implies, of course, that the user is not restricted to only those constructs available in a semantic meta-language. However, it also implies that he must

expend a greater effort to obtain a production compiler. It further implies that the user is able to specify the precise machine configuration on which his language is to be run, again at the expense of time and effort. Input to the XPL system is BNF for the syntactic specification, and XPL code for the interpretive phase.¹

C. DEVELOPMENT

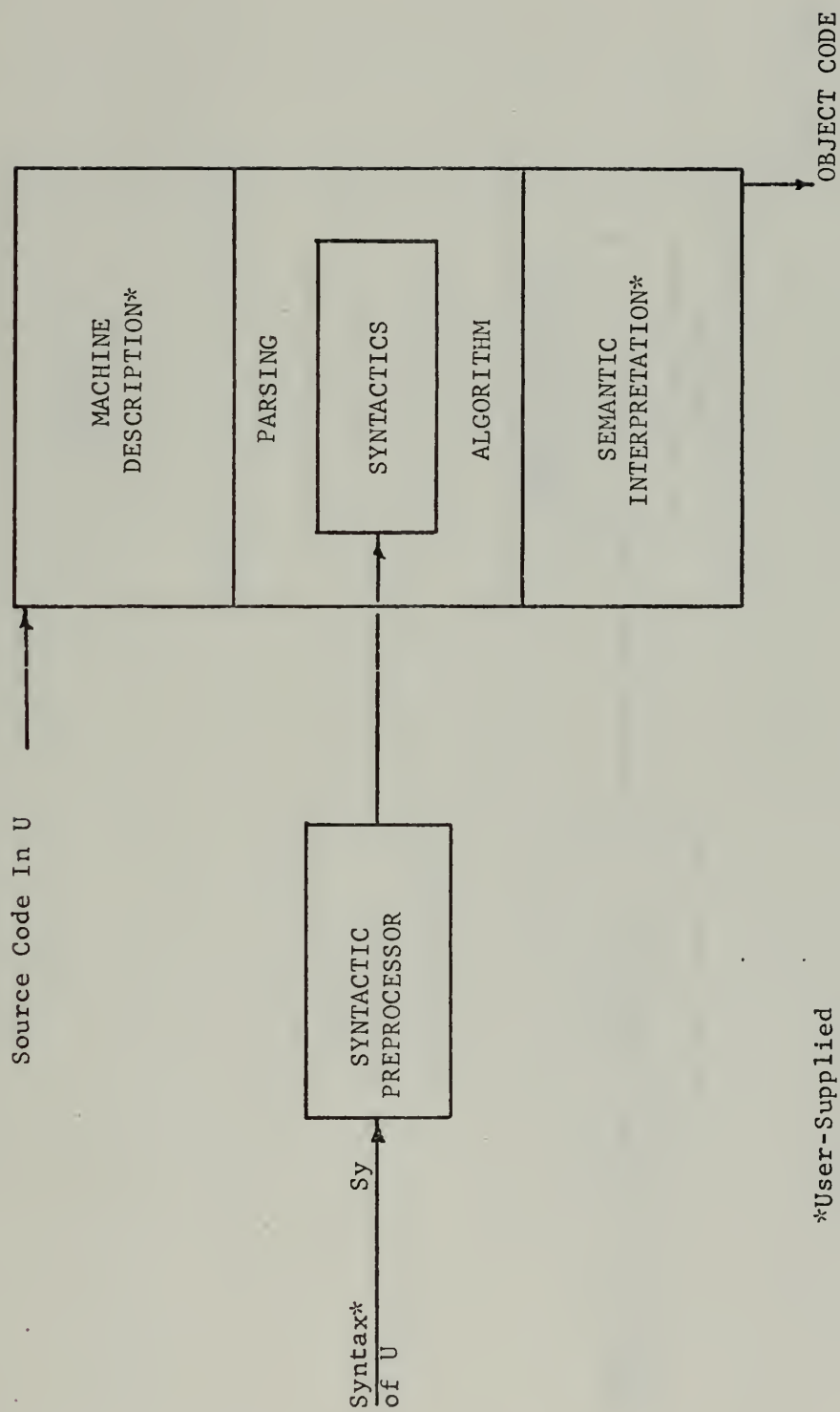
The survey that has been presented is by no means exhaustive. The themes of these efforts are, however, sufficiently representative to allow speculation on the form that a true compiler-compiler might take. Conceptually, Figure 1 represents the CGP in its simplest form. The user encodes the syntax of the user language U in the syntactic meta-language Sy. The syntactic preprocessor accepts Sy as input and produces the syntactics which are inserted into a skeletal parsing algorithm to produce a syntactic analyzer for U. The user must then attach some form of machine description and code production algorithm to the analyzer. Figure 1 represents the XPL system. Figure 2 is identical to Figure 1 except for the automation of the code production algorithm. Now instead of having to hand produce this algorithm, the user encodes the interpretation of the syntactic productions in the semantic meta-language Se. The semantic preprocessor accepts Se as input and produces the semantics which are inserted into a skeletal interpretive algorithm to form a code production algorithm. Figure 2 is representative of the FSL system. Finally, Figure 3 represents a true compiler-compiler. The final difference lies in the addition of a machine preprocessor which accepts a

1. XPL code is a powerful dialect of PL/I; it maintains many of the PL/I constructs and implements many additional features to improve its CGP capabilities.

description of the machine configuration in a mechanical meta-language M and produces the mechanics necessary to complete the CGP. It is an unfortunate fact of life that CGP III has not yet been developed.

The purpose here is to produce a viable CGP in the form of CGP II, based on the XPL system. With that in mind, section II will serve as a short introduction to the XPL system as it presently exists. This will be followed in section III by a complete description of the semantic meta-language, SML.¹ Section IV will report on the implementation of SML and the total XPL CGP package. Finally, section V will conclude with an analysis of this research effort and suggestions for improvements and additions.

1. Prior to reading this section, the reader should acquaint himself with the general format of the XPL CGP by referring to Figure 11.



*User-Supplied

Figure 1

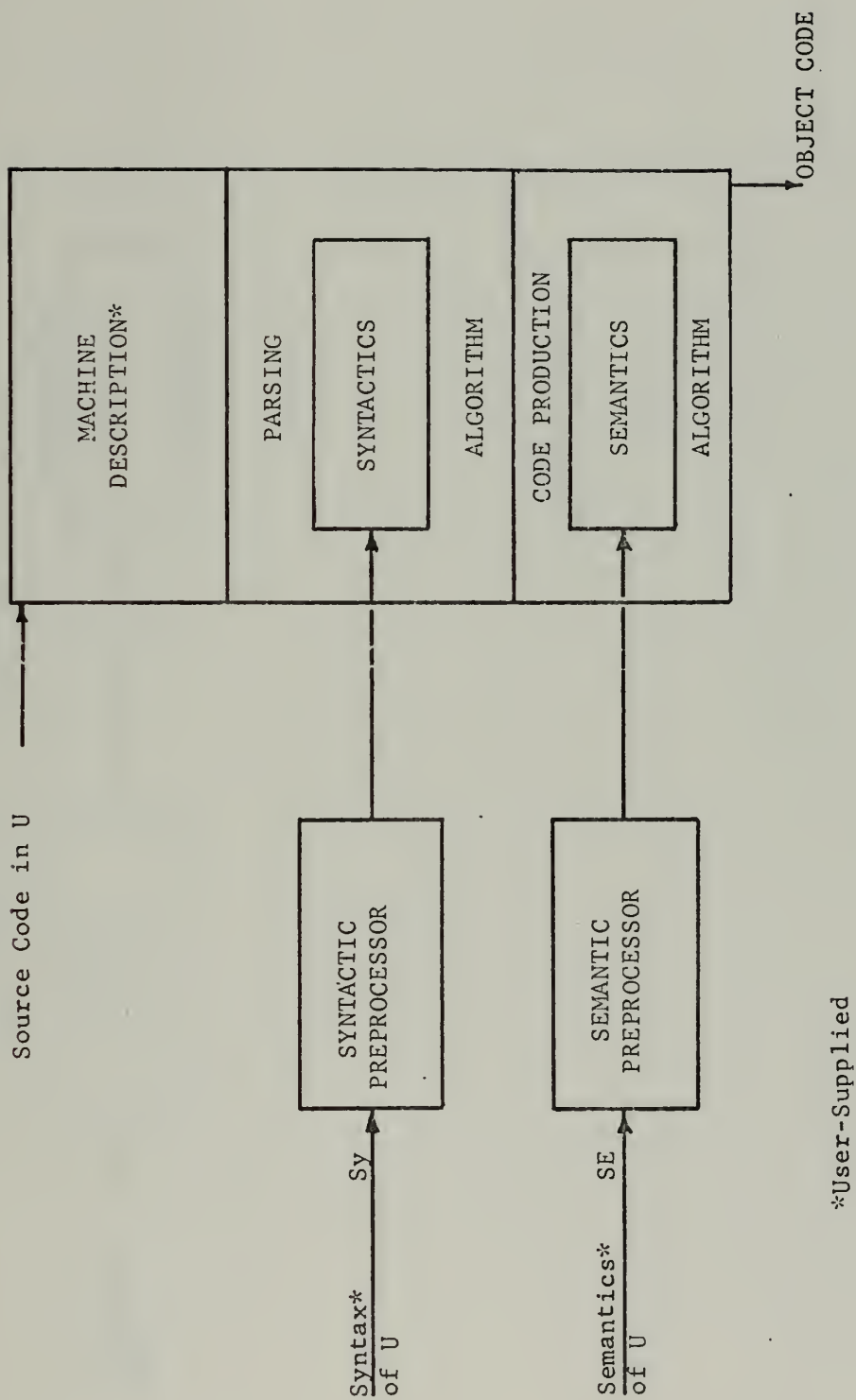


Figure 2

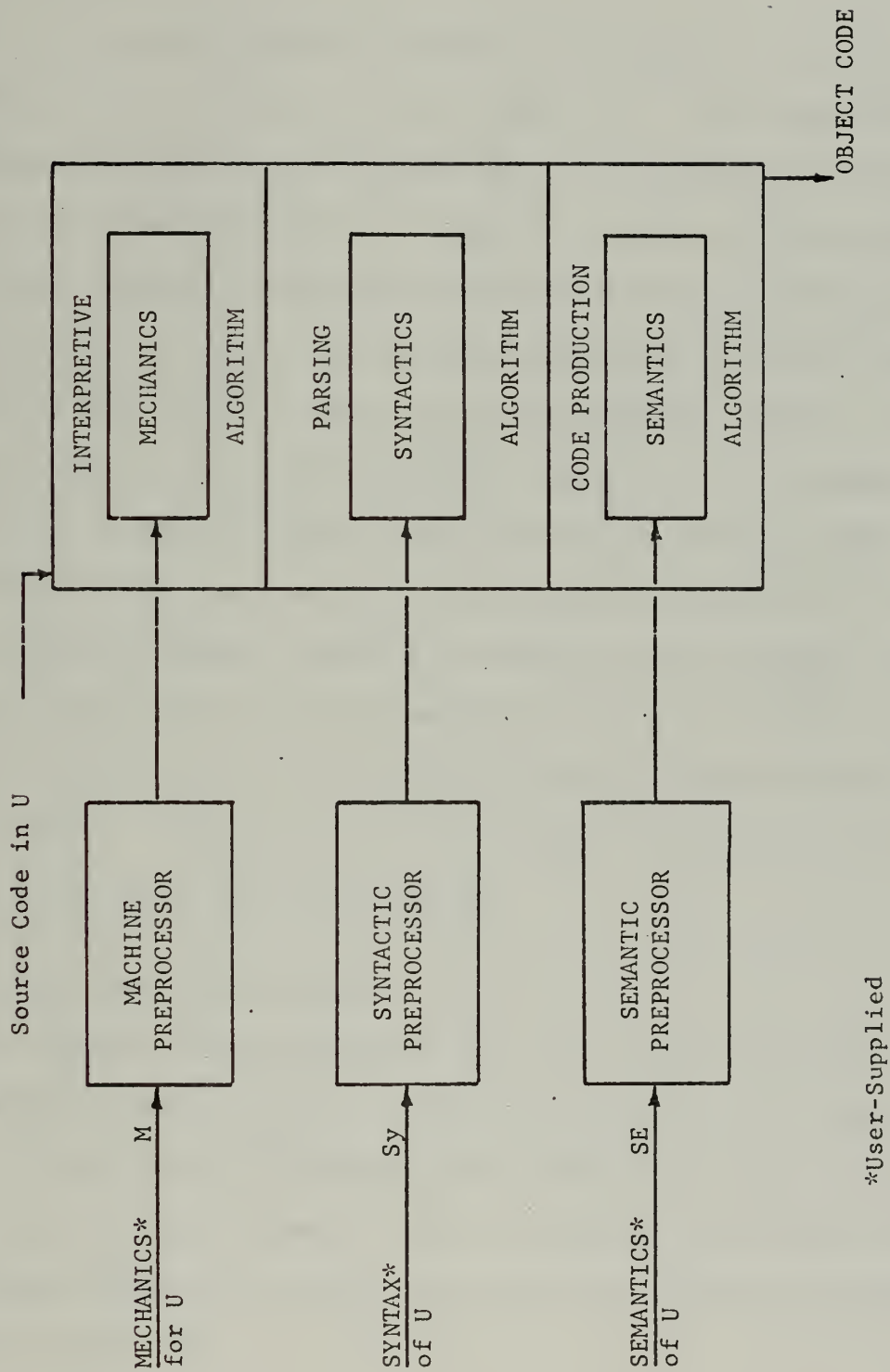


Figure 3

II. THE XPL SYSTEM

A. AN OVERVIEW

The XPL compiler generator system was developed at Stanford by McKeeman et al. in the late 1960's [Ref. 5]. It was specifically designed to be run on an IBM system 360. In its entirety, the XPL system can be viewed as depicted in Figure 4. All batch stream programs, of course, interface with most system 360's by means of IBM Operating System OS/360. In turn, all load modules produced by the XPL system interface with OS/360 by means of the XPL Submonitor (XPLSM). From the user's point of view, source programs encoded in the XPL language are compiled into IBM 360 machine language by the XPL Compiler (XCOM) which is itself an XPL load module run under XPLSM (see Figure 5). Two additional XPL programs, ANALYZER and SKELETON which are also compiled by XCOM and run under XPLSM, essentially complete the XPL CGP. The following sections include a review of the parsing algorithm employed by the XPL system, a closer look at each of the major components of the system, and a brief review of the XPL language.

B. THE XPL PARSING ALGORITHM

The parsing algorithm employed by the XPL system can be broadly characterized as a bottom-up, table-driven procedure. It is often referred to as a mixed-strategy precedence (MSP) algorithm, a name it derives from its ability to handle grammars which are somewhat more complex than simple precedence grammars and somewhat less complex than extended precedence grammars [Ref. 6].

XPL SYSTEM AND ENVIRONMENT

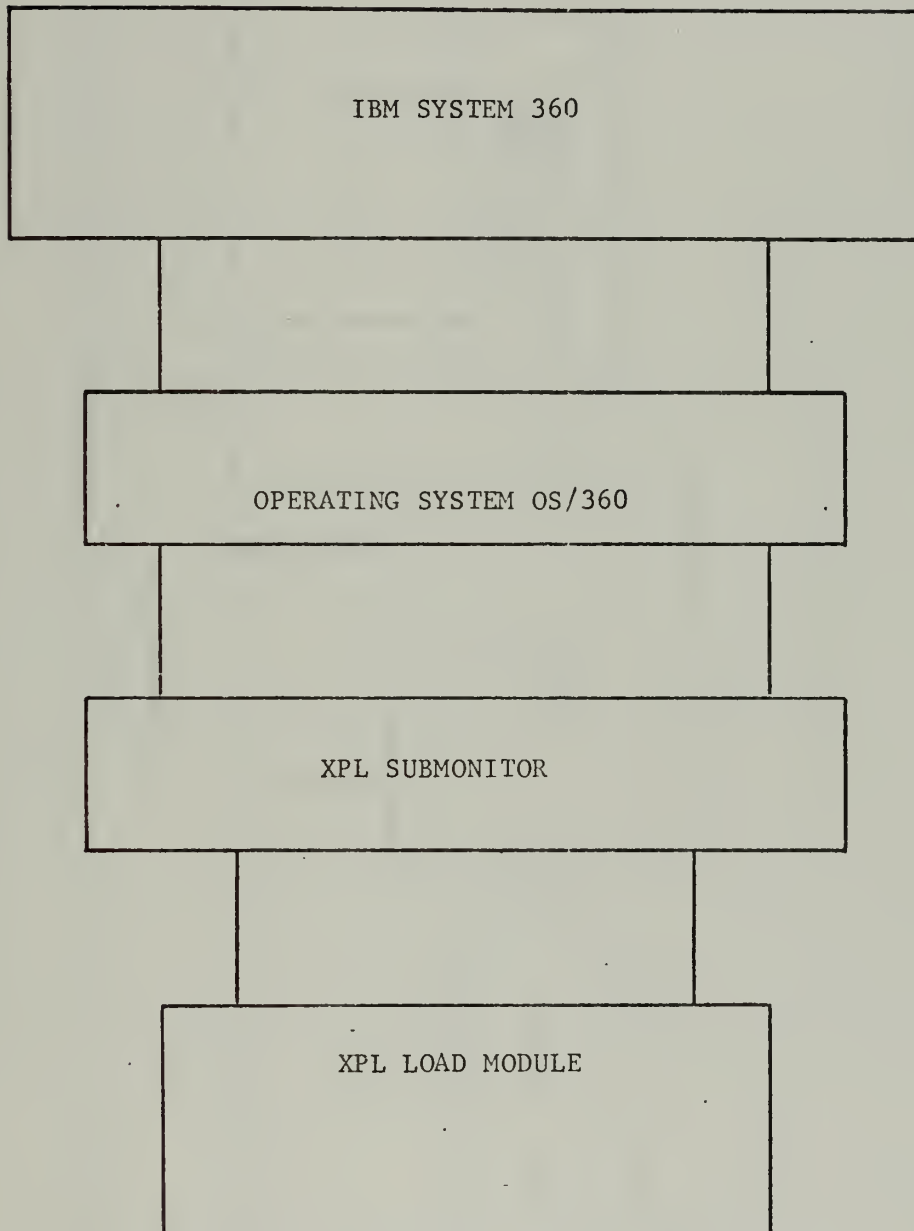


Figure 4

THE XPL ENVIRONMENT FROM THE USER'S VIEWPOINT

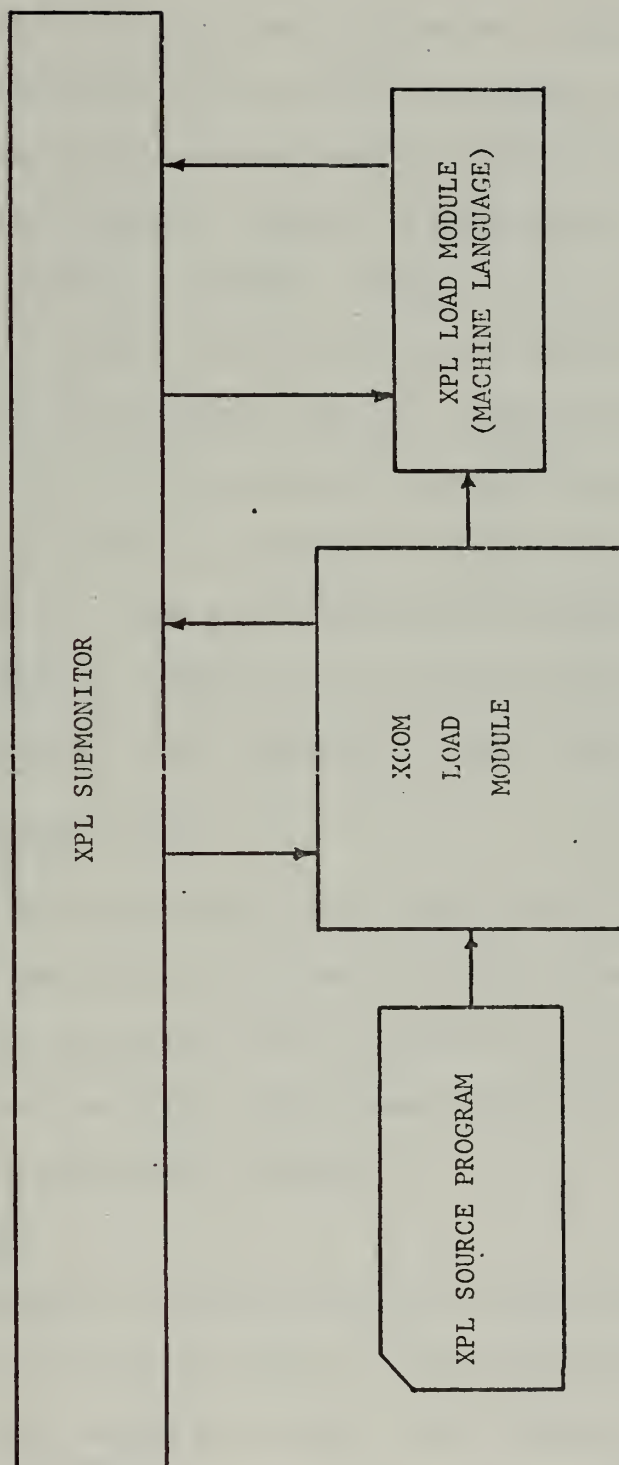


Figure 5

This algorithm is based in the fact that, if a grammar is unambiguous, every sentence has a unique canonical parse. The parsing problem is thus reduced to finding the next canonical parse step at each increment. For the XPL system, this is accomplished by referring to the "canonical parsing function" which is essentially embedded in a set of stacking decision and production selection tables. The essence of the algorithm is as follows: As each new element from the input is picked up, the analysis algorithm determines from the stacking decision tables and the current state of the production stack whether a reduction can be made. If not, the new element is stacked and another is selected from the input. If a reduction is called for, the algorithm determines which production is to be applied by referring to the production selection table. Semantics are associated with syntax immediately before each reduction is made.

C. THE COMPONENTS OF XPL

Figure 6 portrays the XPL CGP in total from the user's point of view. Although the average user will have no direct contact with either XPLSM or XCOM, these two modules form the backbone of the system. On the other hand, all users will require some familiarity with ANALYZER, and nearly total intimacy with SKELETON.

1. XPLSM

Execution of an XPL program under OS/360 begins after OS/360 has loaded XPLSM and given it control. XPLSM then proceeds to open the necessary files, obtain main memory space, and read in the XPL program. Once the XPL program is in memory, XPLSM transfers control to it and it begins execution. During the course of execution, the XPL program may

THE XPL COMPILER-GENERATOR SYSTEM

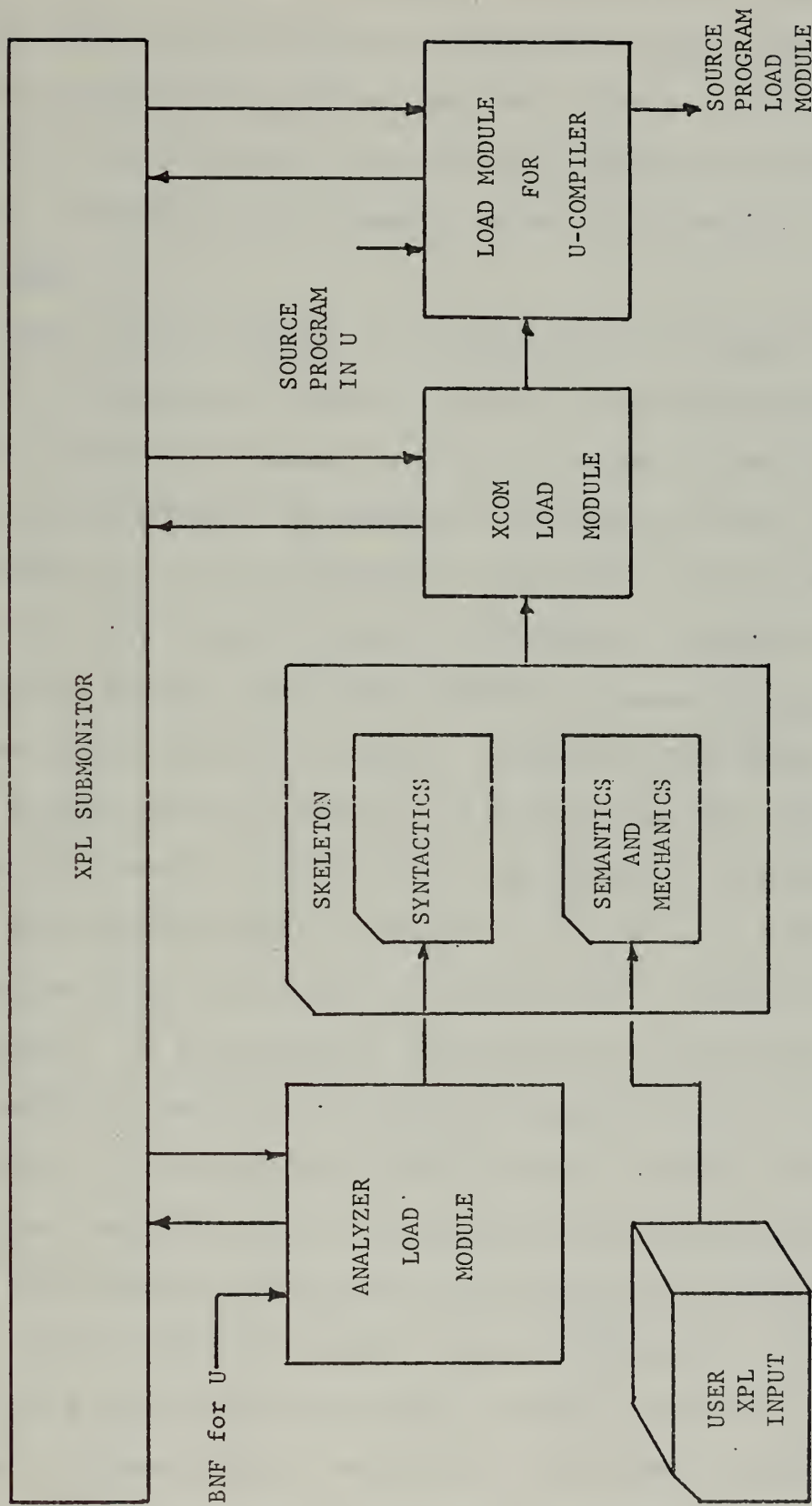


Figure 6

call upon XPLSM with requests for service. In particular, XPLSM has facilities for string input/output, direct access storage, debugging aids, error handling and reporting, and the inclusion of any desirable monitor functions by the user. When the XPL program finally finishes execution, it returns to the submonitor which in turn returns to OS/360.

2. XCOM

XCOM, the XPL compiler, is encoded in the XPL language and is therefore self-compiling. Basically, XCOM is a large XPL program consisting of a sequence of procedures which are invoked in turn by the procedure MAIN-PROCEDURE. The program body consists of only a call to MAIN-PROCEDURE and a return statement to communicate with succeeding steps, if any. The essential procedures of XCOM are displayed in Figure 7. INITIALIZATION tabulates all of the constants and generally lays the groundwork for the compilation process. COMPILATION-LOOP simply invokes a set of procedures which constitute the parsing algorithm. SCAN determines the next symbol in the input text, and STACKING is the precedence function which decides whether a reduction is called for. REDUCE searches the production rules for a match and invokes PR-OK to verify the production selection. If a satisfactory reduction can be accomplished, SYNTHESIZE is invoked to perform various compile-time activities and produce code. Finally, if all goes well, LOADER collates the object code files to facilitate execution of the succeeding step, and PRINT-SUMMARY outputs a set of statistics that have been collected during the compilation.

The principle difference between XCOM and SKELETON lies in the multitude of code emission procedures present in XCOM. This set of procedures, which were hand-coded in XPL, produce an XPL load module containing all of the necessary run-time data structures and some very efficient system 360 machine language code.

PROCEDURES OF XCOM

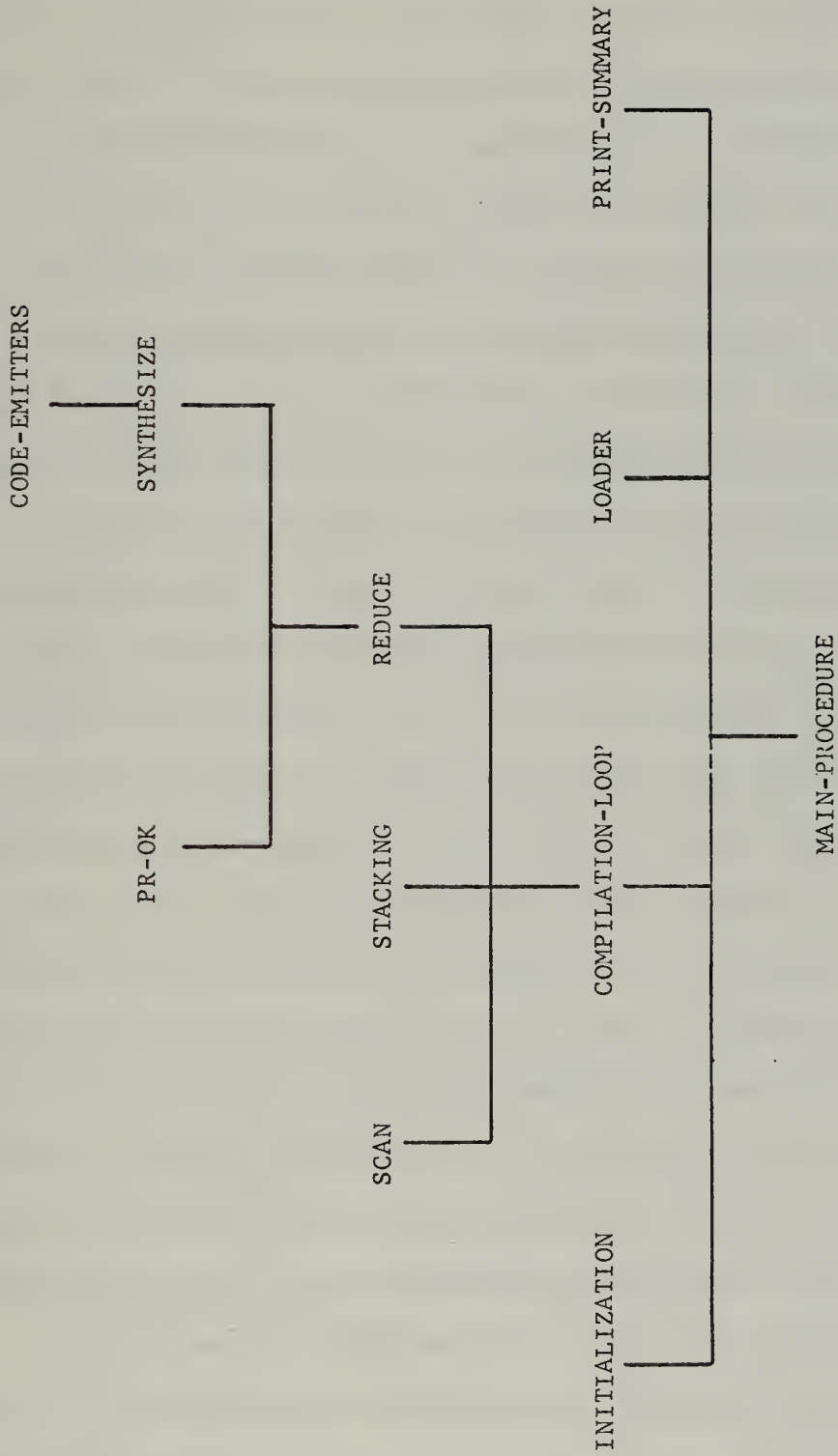


Figure 7

3. ANALYZER

The foremost purpose of ANALYZER is to construct the necessary parsing tables for insertion into the analysis algorithm of SKELETON. From the user's point of view, ANALYZER accepts a BNF description of the user language grammar and produces the required tables on punched cards - assuming there are no errors inherent in either the grammar or in its encoding. Granting that initial efforts to produce an errorless grammar are bound to failure, ANALYZER contains an extensive debugging facility that the user may call up to aid in producing an acceptable MSP grammar.

The most significant procedures of ANALYZER are depicted in Figure 8. The main loop in ANALYZER is contained within the last several lines of the program, which contain a series of calls on the working procedures. The procedure READ-GRAMMAR initializes conditions, calling in turn GET-CARD (reads the next card), SCAN (whose value is the next element of the current card), and SORT-V (which sorts the BNF vocabulary). COMPUTE-HEADS lays some groundwork for succeeding procedures and tests the grammar for symbols that are both left and right recursive. PRODUCE, the heart of the program, invokes APPLY-PRODUCTION and DISAPPLY to simulate recursion, and NEVER-BEEN-HERE to build a table of production contexts. This table is then employed by COMPUTE-C1 to build the stacking decision tables C1 and C1TRIPLES, and further by COMPUTE-C2 to test contexts for possibly troublesome productions. When all of the required tables have been built, PUNCH-PRODUCTIONS punches these tables on cards. If errors have been detected at any stage, ADD-TROUBLE records these errors. If the user has so requested, the procedure IMPROVE-GRAMMAR is invoked after the first iteration in an attempt to resolve the conflicts.

PROCEDURES OF ANALYZER

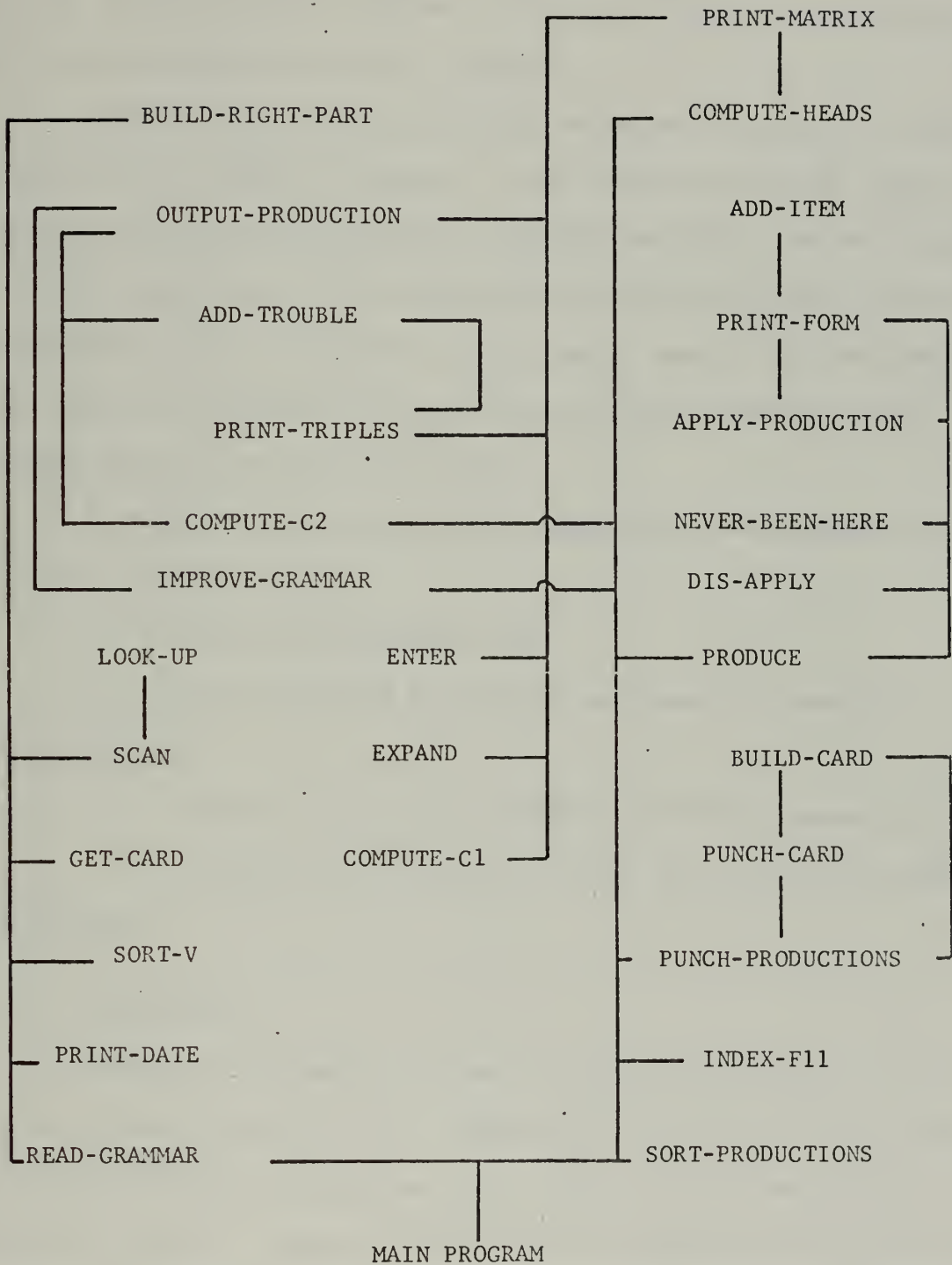


Figure 8

4. SKELETON

SKELETON is the heart of the user's production compiler. Given only the syntactic tables produced by ANALYZER, SKELETON will function as a syntax checker on the user's language.

The major components of SKELETON are shown in Figure 9. (See Appendix A for a complete listing.) As was previously stated, there is little conceptual difference between SKELETON and XCOM. The only significant difference lies in the complete absence of code emitting procedures from SKELETON. It has been left to the user to specify the environment under which his language is to be run, and to attach meaning to the recognition of any specific production.

In brief, to build a production compiler for the desired language the user is required to:

- a. insert the ANALYZER tables;
- b. fine tune SCAN to identify any constructs peculiar to the user language;
- c. specify the working environment of the language; and
- d. construct code emitters and add the required generators to SYNTHESIZE.

D. THE LANGUAGE XPL

The additional programming necessary to convert SKELETON to a production compiler is done in the language XPL. The BNF for XPL is given in Appendix B and a thorough description of all of the constructs is provided by Ref. 5. However, a few of the more important constructs will be briefly presented to enable the reader to follow subsequent programming logic.

PROCEDURES OF SKELETON

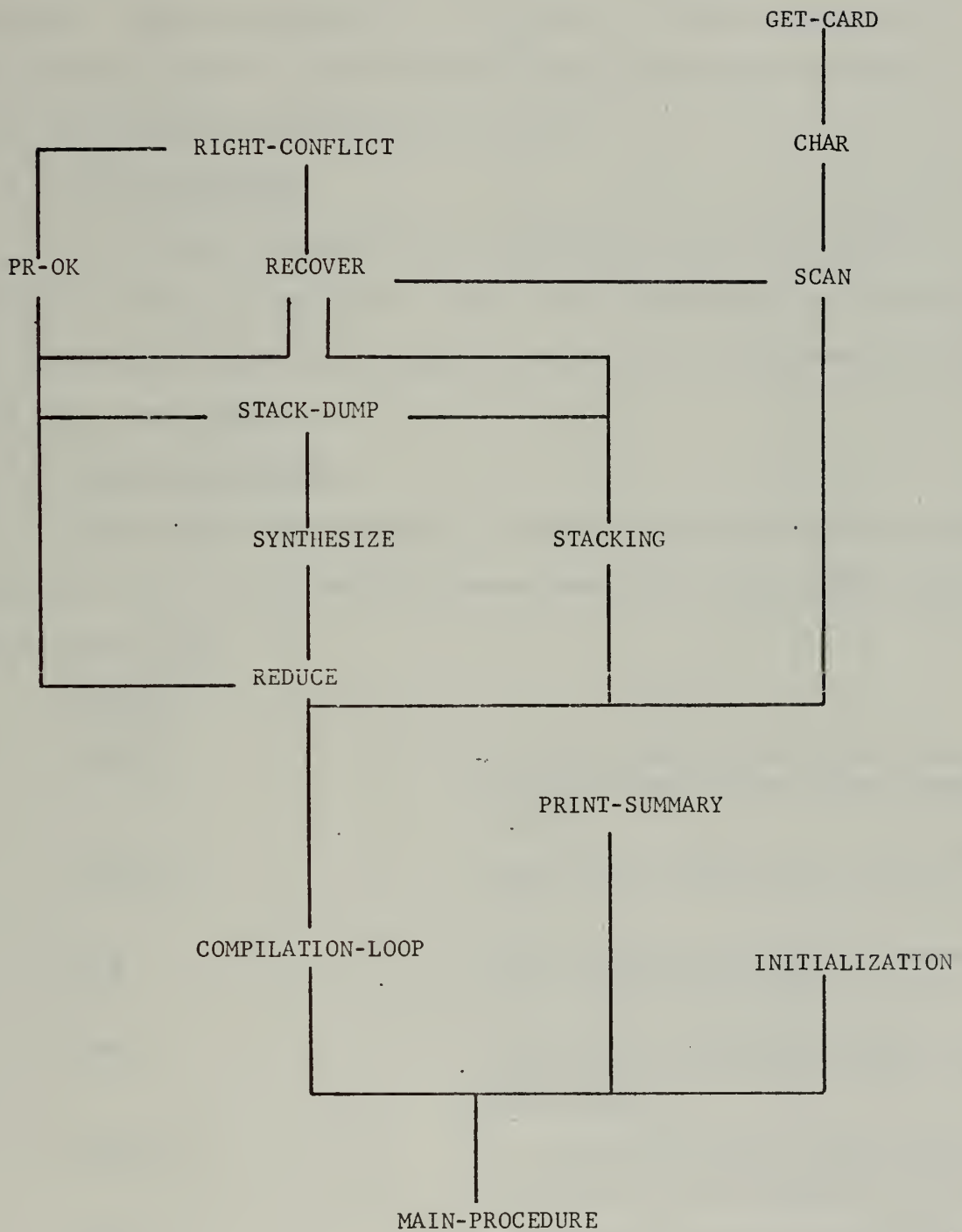


Figure 9

1. MACRO DEFINITION

Simple parameterless macros may be defined using the "literally" attribute. When an identifier is so declared, any later occurrence of that identifier will be replaced during compilation by the character string following the reserved word literally.

2. DO CASE CONSTRUCT

The "DO CASE" construct is a generalization of the selective capability of the "if statement." Thus, the interpretation of the statement "DO CASE I;" would be to cause execution of the Ith statement in the group body that follows.

3. IMPLICIT PROCEDURES

Unless explicitly redefined, a number of procedures are available to the XPL user. Some of the more common ones (e.g. BYTE, SUBSTR, INPUT) are tabulated below:

<u>Function</u>	<u>Use</u>
BYTE(c)	A function with fixed values given by the leading 8 bits of the character string C.
BYTE(c,F)	Like above, except given by the F th 8 bits.
EXIT	A procedure which causes an abnormal exit from XPL execution.
INPUT	A function with character value given by the next record on the input file (SYSIN).
LENGTH(c)	A function with fixed value equal to the length of the character string c.
OUTPUT	An XPL pseudo variable such that any value assigned to it is placed in the output file (SYSPRINT).

FunctionUse

SHL(f1,f2)

A function with fixed value given by shifting the value of f1 left the number of bit positions indicated by f2.

SHR(f1,f2)

The opposite of SHL.

SUBSTR(c,f)

A function with character value given by the substring of c starting at position f to the end of the string.

SUBSTR(c,f1,f2)

Like above except with length f2.

III. THE SEMANTIC META-LANGUAGE

A. AN INTRODUCTION

The XPL CGP is primarily intended to automate the semantic interpretation phase of the XPL system through the use of an intermediate semantic meta-language (SML). Initial efforts to institute SML were largely based on Feldman's work [Ref. 4] . However, SML's development was guided by the XPL environment in which it was to be run.

It was obvious that SML should allow a language to be easily described, both from the standpoint of facility and of time. Furthermore, it should be complete enough and implemented in such a manner as to allow for complete compiler generation insofar as semantics are concerned. Finally, it should be machine independent. To a large extent, these criteria have been met. Where they have not, or where some obviously desirable feature is lacking or could be better implemented, it is reported in the conclusion.

B. THE CONSTRUCTS OF SML

What follows is intended to serve both as a complete description of SML and as a user's guide. The complete BNF for SML is given in Appendix D; and the prospective user is referred to Appendices I-K for detailed examples on using SML.

1. SML Program Structure

The overall structure of a SML program is given by the following

BNF equations:

<semantic program>	::= <header> <sequence> END
<header>	::= <begin> <declaration set>
<begin>	::= BEGIN

<declaration set>	::= <declaration> ; <declaration set> <declaration> ;
<sequence>	::= <bracketed sentence> <sequence> <bracketed sentence>
<bracketed sentence>	::= <delimiter 1> <sentence> <delimiter 2>
<delimiter 1>	::= #
<delimiter 2>	::= @
<sentence>	::= <statement> <sentence> <statement>

Thus a valid SML program, which is interpreted by the Semantic Preprocessor, would be:

```

BEGIN
  <declaration> ;
  <declaration> ;
  @ <sentence> #
  @ <sentence> #
END

```

The important point to extract from this discussion is the use of the delimiters @ and #. Each pair of these delimiters associates a sentence of SML statements with one of the BNF productions. The user, however, is not required to use these delimiters directly. Rather their placement is one of the functions of Prescanner.

2. Declarations

A number of different data structures are available to the XPL CGP user to facilitate compile-time activities:

<declaration>	::= <table declaration> <stack declaration> <cell declaration> <tag declaration> <string declaration> <flag declaration>
---------------	---

For each of these declarations, the Semantic Preprocessor sets up an associated data structure in the basic user compiler, Superskeleton, which is addressable by the given identifier.

a. Tables

Tables are one of the most important structures in a compiler, and are normally used to store a variety of information about run-time structures. The BNF equations for a table declaration are as follows:

```
<table declaration>      ::= <table> <stable specifier>
                           |   <table declaration>
                               <stable specifier>
<table>                  ::= TABLE
<stable specifier>       ::= <stable header> <fields>
<stable header>          ::= <identifier> <number part>
<number part>            ::= ( <number> :
                           |   ( $ :
<fields>                 ::= <field>
                           |   <fields> , <field>
<field>                  ::= <identifier> <type specifier>
<type specifier>         ::= <bit>
                           |   CHAR
                           |   INTEGER
<bit>                    ::= BIT
                           |   BIT <number>
```

An example of a valid table declaration would be:

```
TABLE T1 (100: TYPE BIT 8, LOCATION INTEGER),
        T2 ($: AGE BIT, CITY CHAR);
```

which would be interpreted by the Semantic Preprocessor as a command to set up two tables called T1 and T2 in the user compiler. Table T1 is to have three fields for each of a maximum of 101 entries: a character field for the run-time structure name, a bit field of width 8 to be addressable as TYPE, and an integer field to be addressable as LOCATION. Table T2 is also to have three fields for each entry, but an indefinite number of entries. Indefinite is implied by the \$. This implies in turn that the maximum number of entries is limited only by the capacity of the Super-skeleton table structures. The first field of T2 is also a character field for the structure name. The second is a bit field (of width 32 by default) addressable as AGE, and the third a character field addressable as CITY. The resulting tables can be conceptually viewed as depicted in Figure 10.

THE USER'S VIEW OF TABLES AND STACKS

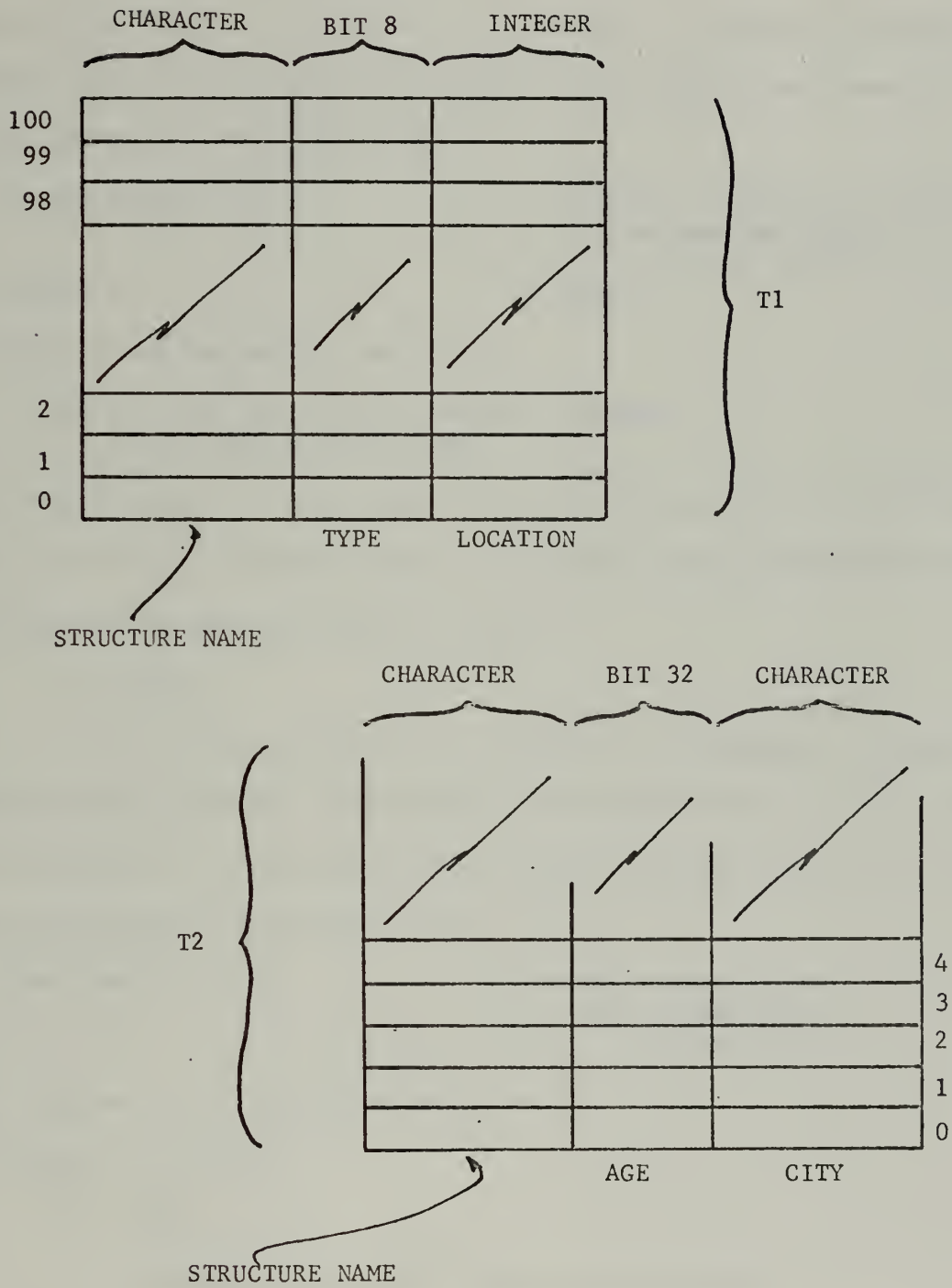


Figure 10

b. Stacks

Stacks are another widely used structure in compiler writing, often being employed for dealing with recursive or temporary constructs. The BNF equations for stack declarations are identical to those for tables except for the first three:

```
<stack declaration>      ::= <stack> <stable specifier>
                           | <stack declaration> ,
                           <stable specifier>
<stack>                   ::= STACK
```

A typical stack declaration might be:

```
STACK S1 (100: TYPE BIT 8, LOCATION INTEGER),
        S2 ($: AGE BIT, CITY CHAR);
```

which would result in user-compiler data structures identical to those for tables T1 and T2 respectively. Of course, S1 and S2 would be treated as and act like stacks instead of tables.

c. Cells

The cell declaration is intended to accommodate the need for single-valued variables. Accordingly, the declaration of a cell causes the allocation of a single word which may be assigned an integer value. The corresponding BNF equations are:

```
<cell declaration>      ::= CELL <identifier>
                           | <cell declaration> ,
                           <identifier>
```

and a typical cell declaration would be:

```
CELL I1, I2, I3, I4;
```

d. Tags

The BNF equations for a tag declaration are:

```
<tag declaration>      ::= TAG <identifier>
                           | <tag declaration> , <identifier>
```

The tag construct can be useful in "tagging" fields of a table or stack

as having certain characteristics. For instance, give the declarations:

```
TABLE T1 ($: TYPE BIT 6,LOC CHAR);  
TAG MATRIX,VECTOR,CONSTANT,STRING;
```

then the field TYPE might be tagged as either a MATRIX, VECTOR, CONSTANT, or STRING by an appropriate assignment statement.¹

The interpretation of a tag declaration by the Semantic Preprocessor causes a unique XPL literal declaration to be made in the user compiler.

e. Strings

To facilitate the manipulation of strings in the user compiler, string variables may be declared according to the following BNF equations:

```
<string declaration>      ::= STRING <identifier>  
                           |      <string declaration> ,  
                           |      <identifier>
```

A typical string declaration such as:

```
STRING ST1;
```

would cause the allocation of a character string addressable as ST1.

f. Flags

The BNF equations for a flag declaration are:

```
<flag declaration>        ::= FLAG <identifier>  
                           | <flag declaration> ,  
                           | <identifier>
```

Flags are essentially simple Boolean variables which may take on the value either TRUE or FALSE. They are often useful to specify the state of the compilation process.

Given the declaration:

```
FLAG F1;
```

the Semantic Preprocessor would allocate, in the user compiler, a flag

1. Note that a bit field of width 6 is always sufficient to utilize the tag construct.

addressable as F1 and initialized to the value FALSE.

3. SML Variables

Before considering the various operations that can be performed upon the data structures described above, a brief overview of the SML primitive variables is necessary. There are basically two classes of variables, boolean and arithmetic.

a. Boolean Variables

The BNF equations for Boolean variables are as follows:

<boolean>	::= <boolean term>
	<boolean> <or> <boolean term>
<or>	::= OR
<boolean term>	::= <boolean factor>
	<boolean term> <and>
	<boolean factor>
<and>	::= AND
<boolean factor>	::= <boolean primary>
	<not> <boolean primary>
<not>	::= NOT
<boolean primary>	::= <arithmetic> <relation>
	<arithmetic>
	<lrel*> <boolean> <rrel*>
	<test> <primary>
	TRUE
	FALSE
<test>	::= TEST
<lrel*>	::= <
<rrel*>	::= >
<relation>	::= .LSS.
	.LEQ.
	.EQL.
	.NEQ.
	.GEQ.
	.GTR.

All of these constructs, except the use of <lrel*> <boolean> <rrel*> and <test> <primary>, have the usual interpretation. The <lrel*> <boolean> <rrel*> merely indicates grouping, but the test construct requires more explanation.

In brief, the test construct can be employed to extract information about the state of various data structures. Given the

boolean primary "TEST A", then if A is a cell, TEST A will have the value TRUE if A is greater than zero. If A is a string, then TEST A will have the value TRUE if A is the null string. If A is a flag, then TEST A will have the value TRUE if A is true. Finally, if A is a stack, then TEST A will have the value TRUE only if A is not empty. In all other cases, TEST A will have the value FALSE.

b. Arithmetic Variables

For arithmetic variables, the BNF equations are:

```

<arithmetic>          ::= <term>
                        | <+> <term>
                        | <-> <term>
                        | <arithmetic> <+> <term>
                        | <arithmetic> <-> <term>
<+>                   ::= +
<->                   ::= -
<term>                 ::= <factor>
                        | <term> <*> <factor>
                        | <term> </> <factor>
<*>                   ::= *
</>                   ::= /
<factor>               ::= <primary>
                        | <factor> <*> <primary>
<*>                   ::= **
<primary>              ::= <operand>
                        | <number>
                        | <bit string>
                        | <character string>
                        | <ldel*> <arithmetic> <rdel*>
<operand>              ::= <stable operand>
                        | <floating operand>
                        | <storage operand>
                        | <code operand>
                        | ( <list operand> )
<storage operand>      ::= <identifier>
<list operand>         ::= <primary> ; <primary>
                        | <list operand> ; <primary>

```

Most of these constructs have their usual meaning. As with boolean variables, the use of <ldel*> <arithmetic> <rdel*> is to indicate grouping. The list operand, which allows for list assignments of up to

16 variables, is one of the more useful constructs in the compiler writing process.¹ Discussion of the stable operand, floating operand, and code operand is deferred to the appropriate sections.

4. SML Operations

Before proceeding with a discussion of SML operations on data structures, it is important to note that SML primitives cannot be embedded in one another. This perhaps unfortunate restriction is due to the way in which the Semantic Preprocessor incrementally examines the SML input and constructs the XPL code stream output.

The BNF equations for general statement types are as follows:

<statement>	::= <unconditional> ; <conditional> <label> <statement> <comment>
<comment>	::= <lidel> <remarks> <rdel>
<lidel>	::= /*
<rdel>	::= */
<label>	::= <identifier> :
<unconditional>	::= <assignment> <operation> <transfer>
<operation>	::= <table operation> <stack operation> <code operation> <auxiliary operation>
<transfer>	::= JUMP <identifier>

The only terminal constructs introduced in the above equations are the statement label and comment. The appearance of a comment, for example:

/* THIS IS A COMMENT */

is given no semantic interpretation and hence is ignored by the Semantic Preprocessor. Neither does the label construct have any meaning. It

1. The interpretation of the list assignment: (A;B):=(C;D); would be: A:=B; and C:=D;.

does, however facilitate transfer of control within the SML program. For example:

```
BEGIN
  -----
  LAB:  -----
  -----
  JUMP LAB;
  -----
END
```

When the JUMP LAB statement is encountered in the SML program, control is transferred to the statement labelled LAB.

The discussion of the various SML operations which follows is separated into sections according to the data structures involved.

a. Assignment Statement

The most general SML operation is the assignment statement which can be used with all types of data structures. The appropriate BNF equations are:

```
<assignment>      ::= <left part> <arithmetic>
                   |   <left part> <boolean>
<left part>        ::= <primary>   :=
```

and they have the usual interpretation. Considering only the primitive variables, the following are valid examples of an assignment:

```
BEGIN
  CELL A,B,C,X,Y,Z;
  STRING ST1,ST2,ST3;
  FLAG F1,F2,F3;
  -----
  A  := X;
  B  := 720;
  (A;B;C) := (X;Y;Z);
  C  := A* <B+C/ <72-< X**Z>>>;
  ST1 := ST2;
  ST3 := 'ABCXYZ';
  F1 := "1";
  F2 := TRUE;
  F3 := TEST A;
  F1 := NOT TEST A;
  F2 := <A+B> .LSS. <X-Y> ;
  -----
END
```


b. Cell Operations

There are only two commands which deal with cells, the BNF equations for which are:

```
<auxiliary operation> ::= TALLY <identifier>
                        | MINUS <identifier>
```

If A is a cell, then the statements:

```
TALLY A;
MINUS A;
```

are equivalent to

```
A := A+1;
A := A-1;
```

respectively.

c. Table Operations

There are a number of SML constructs available to facilitate manipulating tables. The declarations:

```
TABLE T1(100:TYPE BIT 6,LOC BIT),
      T2($:ATYPE BIT 6,ALOC BIT);
CELL X,Y,Z;
TAG MATRIX,VECTOR,CONSTANT;
```

will be used in the discussion which follows.

The enter command is probably the most useful table command.

The corresponding BNF equations are:

```
<table operation> ::= <enter operation>
<enter operation> ::= <entry point1> <list operand> )
                    | <entry point1> <primary> )
<entry point1>    ::= <entry point>
<entry point>     ::= <enter> <variable> :
<enter>           ::= (ENTER) <identifier>
<variable>        ::= = <primary>
```

Thus, a typical enter command might be:

```
ENTER (T1 = 'NAME':MATRIX;X);
```

which would cause the ordered triple ('NAME',MATRIX,X) to be entered into the table data structure addressed as T1. The only restriction on the

use of this command is that all fields as given in the corresponding declaration must be accounted for. Hence the command:

```
ENTER (T1 = 'NAME':MATRIX);
```

would not be valid. The enter command could still be used in this situation by the addition of a dummy variable; for instance:

```
ENTER (T1 = 'NAME':MATRIX;0);
```

would be valid.

In order to locate a specified entry in a particular table, the loc operand can be used in conjunction with an assignment statement. The BNF equations for this construct are:

<stable operand>	::= <loc operand>)
<loc operand>	::= <loc> <variable>
<loc>	::= LOC (<identifier>

and a typical locate command might be:

```
A := LOC (T1 = 'NAME');
```

The execution of this statement would cause the index of the entry 'NAME' in table T1 to be assigned to cell A. This index can be used in conjunction with two more table operations, the delete and set operations.

The delete command, which has two forms, can be used to delete a specified entry in a particular table. BNF equations for the delete operation are:

<table operation>	::= <delete operation>
<delete operation>	::= <delete> <variable>)
	DELETE <identifier>
<delete>	::= DELETE (<identifier>

If the index of the entry is known, the short form should be used for efficiency. For example:

```
A := LOC (T1 = 'NAME');  
-----  
DELETE A;
```


is the most efficient method for removing the entry associated with the structure 'NAME'. If, however, the index is unknown then the other form of the delete command will cause the same action; for instance:

```
DELETE (T1 = 'NAME');
```

will have the same effect as the first form.

If only the structure name is known and the user wishes to allocate a table entry to be manipulated at a later time, then the most appropriate method would be to use the entry operand, the BNF equations for which are:

```

<stable operand>      ::= <entry operand> )
<entry operand>      ::= <entry> <variable>
<entry>               ::= ENTRY ( <identifier>

```

An example of its use would be:

```
B := ENTRY(T2 = 'NAME');
```

which would cause the entry of the structure name 'NAME' into table T2, the reservation of fields ATYPE and ALOC to be associated with 'NAME', and the assignment of the index of the entry to B. As with the loc operand, this index can be used with either the delete or set operation.

The final type of table operation employs the set operand, the BNF equations for which are:

```

<stable operand>      ::= <set operand> )
<set operand>         ::= <set point> <list operand>
                        <set point> <primary>
<set point>           ::= <set point 1> <variable> :
                        <set point 1> :
<set point 1>         ::= <set> <identifier>
<set>                 ::= SET (

```

Like the delete operand, the set operand has two forms depending upon whether the entry index is known. The first form, an example of which might be:

```
(A;B) := SET(T1 = 'NAME':TYPE;LOC);
```


is used when the index is unknown and only to extract table information. In this particular example, the information contained in the TYPE field of the entry associated with the structure 'NAME' is assigned to A; the information in the corresponding LOC field is assigned to B. When the index is known, the set operand may appear on either side of the assignment statement. For instance,

```
A := SET(B:TYPE);
```

is interpreted as

```
A := TYPE(B);
```

and

```
SET(B:TYPE) := A;
```

is interpreted as

```
TYPE(B) := A;
```

It is important to note that, in any case, the identifiers in the set operand specify the field names of the appropriate table; they are not variables.

d. Stack Operations

There are a number of SML constructs available for manipulating stacks. In the discussion which follows, the following declarations will be referred to:

```
STACK S1(100:TYPE BIT,LOC INTEGER),  
      S2($:ATYPE BIT,ALOC INTEGER);  
CELL A,B;  
STRING X,Y;  
TAG UP,DOWN;
```

The simplest form of a stack operation is an assignment statement of the form:

```
SL := S2;
```

which will cause the field contents of S2 to be "pushed" onto the

corresponding fields of S1, and S2 to be "popped." The obvious restriction in the use of this operation is that the number and attributes of the stack fields be identical.

To accomplish exactly the same purpose as described above, except that S2 is not to be popped, the top operand can be used. The corresponding BNF equation is:

`<stable operand> ::= TOP <identifier>`

and an example of its use would be:

`S1 := TOP S2;`

An entire set of entries can be pushed onto a stack by using a list operand in conjunction with an assignment statement, for example:

`S2 := (X;UP;A);`

The reciprocal of this type of an assignment can be used to extract the respective field contents and pop the stack:

`(X;A;B) := S2;`

To extract the field contents, but not pop the stack, the top operand can be used as follows:

`(X;A;B) := TOP S2;`

In all of the above cases where the list operand is employed, it is important to note that the number and type of primitive variables in the list should be identical to the fields of the corresponding stack.

To manipulate specific entries in the stack, the set operand may be used. For instance, to enter a structure name and a single field, the command:

`SET(S1 = 'NAME':TYPE) := DOWN;`

might be used. To fill the LOC field at a later time, use the command:

`SET(S1:LOC) := A;`

Or, to extract information from a specific field:

```
A := SET(S2:ATYPE);
```

In summary, the set operand can be used to manipulate stacks exactly as to manipulate tables, with the exception of using the stack name as an index for stacks instead of an actual index as for tables.

Finally, to simply pop a stack the pop command may be used.

The corresponding BNF equation is:

```
<stack operation> ::= POP <identifier>
```

and an example of its use would be:

```
POP S1;
```

e. Code Operations

SML code is interpreted by the Semantic Preprocessor and translated into an XPL code stream which is one of the inputs to the basic user compiler, Superskeleton. The completed user compiler subsequently interprets the user language code and translates it into Balgol-2 machine code, which is executed by the stack-oriented machine simulator, Balgol-2 [Ref. 7].

To insert XPL code directly into the user compiler or to cause the user compiler to emit machine code, the direct and indirect code commands may be employed. The appropriate BNF equations are:

```
<code operation> ::= NOCODE
                  | <code>
<code> ::= <direct>
          | <indirect>
<direct> ::= CODE <character string>
<indirect> ::= <code 1> <character string> )
              | <code 1> <operand pair> )
              | <code 1> <contents> )
<code 1> ::= CODE
<operand pair> ::= <operator>
                  | <operator> , <character string>
<operator> ::= OP <character string>
<contents> ::= CONT <character string>
```


The nocode command, simply

```
NOCODE;
```

is equivalent to the standard NOP instruction and has no interpretation in either the Semantic Preprocessor or the user compiler.

In order to insert XPL code into the user compiler code stream, the direct form of the code command is used. An example of this command would be:

```
CODE 'FIXV(MP) = LSS;';
```

This command will prove useful in manipulating the parse stack entries in the user compiler, and in handling situations not otherwise covered by SML.

In order to cause the user compiler to emit machine code, in turn causing a value to be loaded into the Balgol-2 execution stack, the first form of the indirect code command is used. For example:

```
CODE ('72');
```

will cause the number 72 to be loaded into the execution stack at run time. The command:

```
CODE ('B');
```

will likewise cause the contents of the variable addressed as B to be loaded into the execution stack.

To cause the user compiler to emit a machine command or an operator/operand pair, the second form of the indirect code may be used. For instance:

```
CODE (OP 'ADD');
```

will cause the insertion of the ADD operator into the machine code stream. And:

```
CODE (OP 'IM1', 'FIXV(SP)');
```


will cause the insertion of the IML operator and the value of the variable FIXV(SP) into the machine code stream.

To provide for a level of indirect addressing, the third form of the indirect code command is used. Thus the command:

```
CODE (CONT 'B');
```

will cause the value of the variable whose address is contained in the variable B to be loaded into the execution stack.

f. Statement Control

The execution of SML statements can be controlled with either the transfer operation covered earlier or the conditional statement, the BNF equations for which are:

<conditional>	::= <if clause> <then clause> <if clause> <then clause> <else clause>
<if clause>	::= <if> <boolean>
<if>	::= IF
<then clause>	::= <thendo> <sentence> END;
<else clause>	::= <elsedo> <sentence> END;
<elsedo>	::= ELSEDO;
<thendo>	::= THENDO;

The conditional statement has the same interpretation as in XPL or PL/I. A valid example might be:

```
If TEST S1 THENDO;  
  CODE 'A=B;';F1:=FALSE;  
END; ELSEDO;  
  CODE 'A=C;';F1:=TRUE;  
END;
```

The execution of run-time statements can be controlled with the FLOP/CODELOC construct described in the BNF equations:

<floating operand>	::= <flop> <primary>) <flop> <program label>)
<flop>	::= FLOP
<program label>	::= ADDR <primary>
<code operand>	::= CODELOC

In conjunction with the assignment statement, there are four possible interpretations which are made depending on the statement form. A statement of the form:

```
FLOP(ADDR 'VAR(MP)') := CODELOC;
```

would mean that the present code location should be saved and associated with the value of the variable VAR(MP). The inverse statement:

```
CODELOC := FLOP(ADDR 'VAR(MP)');
```

would mean that the code location associated with the value of VAR(MP) should be inserted into the machine code stream. This particular construct lends itself well to the implementation of such language structures as statement labels and do loops (see the sample language).

On the other hand, a statement of the form:

```
FLOP ('ELSE') := CODELOC;
```

would mean that a branch to a presently unknown code stream location is to be inserted at this point, which is identified by 'else'. The appearance of the inverse:

```
CODELOC := FLOP ('ELSE');
```

would cause the previously reserved branch command associated with the identifier 'else' to be made to the current code stream location. The use of these two statements can easily facilitate the implementation of such structures as conditional statements. It is important to note that the implementation of this statement control construct allows embedding of structures up to 32 levels deep by associating only the latest occurrence of the flop identifier with the backstuffing request.

g. Miscellaneous Statements

The SML error command can be used to cause the user compiler to output error messages. The appropriate BNF equations are:

<auxiliary operation>	::= <error> <primary>
<error>	::= ERROR

A possible use of this command might be something on the order of:

```
IF NOT TEST S1 THENDO;
  ERROR 'STACK IS EMPTY';
END;
```

In order to stop the compilation process in the user compiler, the stop command can be issued. The associated BNF equation is:

<auxiliary operation>	::= STOP
-----------------------	----------

and a typical use might be:

```
IF TEST ST1 THENDO;
  STOP;
END;
```


IV. THE XPL CGP

A. AN OVERVIEW

In its final form, the XPL CGP is a complete compiler generator system for the class of MSP languages and the Balgol-2 interpretive machine. It is composed of five distinct but closely interrelated modules: PRESCANNER, Syntactic Preprocessor, Semantic Preprocessor, Superskeleton, and the Balgol-2 Machine. Figure 11 shows how these modules are related to one another and the XPL system in which they are based.

B. SYSTEM INPUT

The only system input necessary to completely specify the user compiler in many cases will be the BNF and SML for the user language. The input package has the following general outline:

```
control cards
  BEGIN
    declaration set
    BNF and associate SML
  END
EOF
```

1. Control Cards

Within both Syntactic Preprocessor and Semantic Preprocessor there are a number of control toggles which can be turned on to initiate various trace and debugging routines. To complement a specific toggle, the user must input the appropriate control card. The general format of a control card is \$SYNTAX. or \$SEMANTICS. punched beginning in card column 1 and followed by the appropriate keyword. The functions which are toggle controlled and the initial toggle states are as follows:

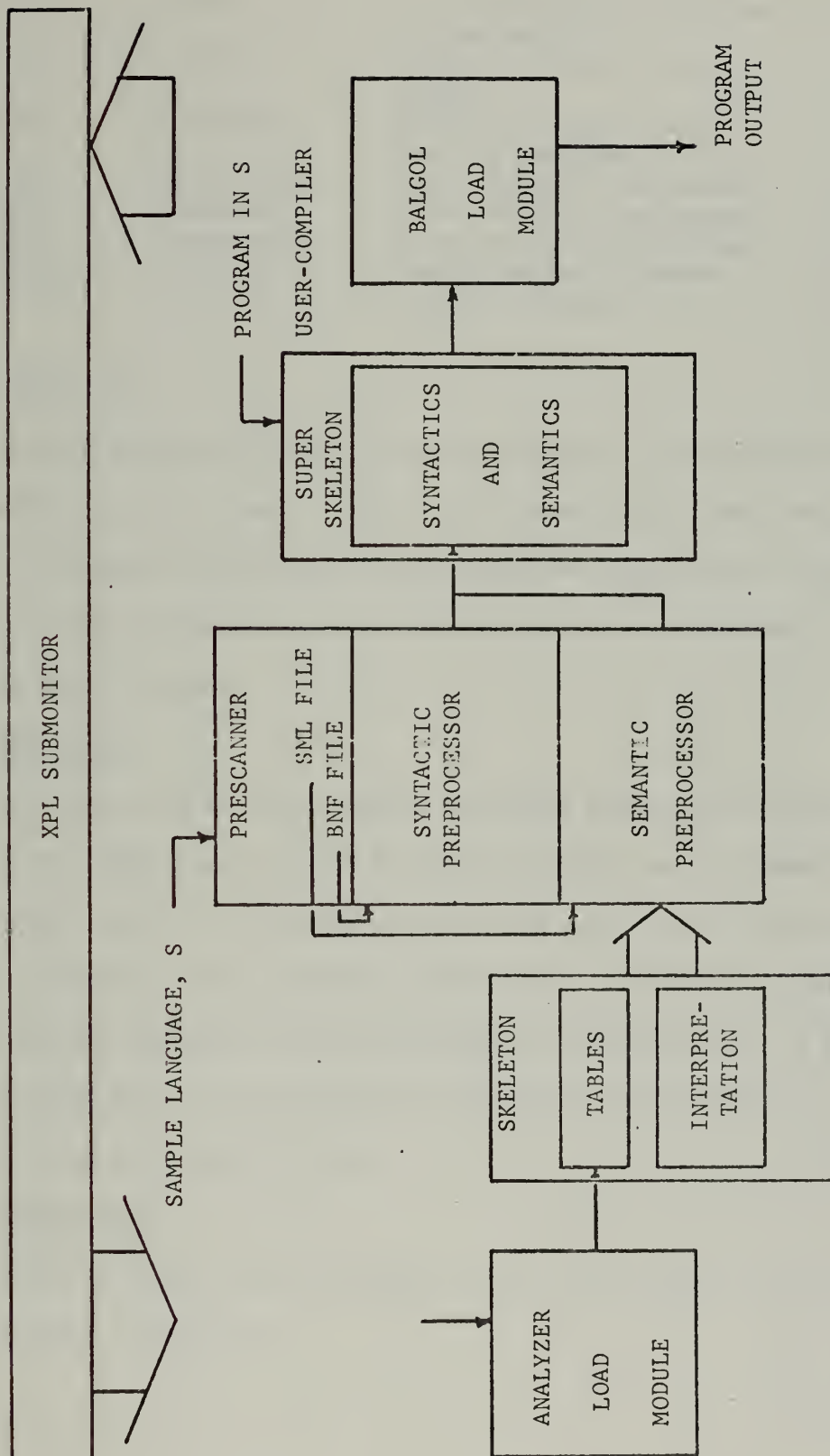


Figure 11

<u>keyword</u>	<u>type</u>	<u>use</u>	<u>state</u>
LIST	both	list the analysis program on the printer	on
PUNCH	both	produce punched card output	off
TRACE	\$syntax.	produce a machine trace for the program	off
CODE	\$semantics.	list the XPL code output	on
OUTPUT	\$syntax.	list the XPL code output	off
DUMP	\$semantics.	print the run statistics	off
ITERATE	\$syntax.	try to correct grammar errors, if any	off

2. BNF Input

The BNF productions for the user language are encoded in the normal manner, one to a card. The primary production (the first of a series of alternate definitions) must be punched beginning in card column 1. All alternate productions must be preceded by an asterisk, also punched in card column 1.

3. SML Input

All SML cards must be punched beginning after card column 2, including the BEGIN card and all declarations which are to immediately precede the first BNF production, and the END card which follows the last SML statement. The important thing to note is that there must be at least one SML statement for and following each BNF card. If no semantic information is attached to a particular BNF production, then the NOCODE command should be used.

4. End-of-file

The last input card is the end-of-file (EOF) card which is punched in card columns 1-3.

C. PRESCANNER

The function of PRESCANNER is to go through the input deck and create two input files for Syntactic Preprocessor and Semantic Preprocessor respectively. In the process, PRESCANNER performs a number of other important functions. It checks each input card for format errors. It ensures that each BNF production has some semantic interpretation, even if only NOCODE. Finally, it formalizes the association of SML to BNF by the appropriate placement of the SML delimiters @ and #.

D. SYNTACTIC PREPROCESSOR

The algorithm of the XPL procedure ANALYZER has been unaltered in Syntactic Preprocessor. Major changes in format and minor changes in the program have been made only to simplify readability and clarity.

Additionally, the final program statement:

```
RETURN ERROR_COUNT;
```

has been added to make the execution of succeeding steps conditional on the success of Syntactic Preprocessor in producing the syntactics for Superskeleton.

The input to Syntactic Preprocessor is the corresponding file created by PRESCANNER in the previous step. Assuming a \$SYNTAX.PUNCH control card in the CGP input and the successful analysis of the grammar, Syntactic Preprocessor outputs punched cards which are then inserted into Superskeleton.

E. SEMANTIC PREPROCESSOR

Semantic Preprocessor interprets the SML file created by PREPROCESSOR and produces two sets of punched cards (assuming a \$SEMANTICS.PUNCH control card was included in the CGP input and no errors are encountered).

The first is an XPL declaration which sets up the necessary user compiler data structures. The second is an XPL source code stream which will execute the associated semantic interpretation. When these two sets of cards, and those produced by Syntactic Preprocessor, are added to Superskeleton, the result is a complete compiler for the user language.

As depicted in Figure 12, Semantic Preprocessor was largely built from the existing XPL compiler generator system. The BNF for SML was first input to ANALYZER, and the resulting syntactics were added to Skeleton to produce a syntax checker for SML. Then the semantics and mechanics were hand-encoded to interpret the SML program input and output the appropriate declarations and XPL code stream. The final program structure is thus typical of any user compiler generated under the current XPL system.

F. SUPERSKELETON

Superskeleton is the basis of the user compiler. Like Semantic Preprocessor, it had its beginnings in the current XPL system. It is essentially a copy of Skeleton with Balgol-2 mechanics and SML-interpretive data structures added. The Balgol-2 mechanics are straightforward and can be easily understood by referring to Appendices G and H.

Some of the data structures require more than a passing glance. All table entries are made via a hashing scheme and the stack quadruple (KEY, ID, OVERFLOW, POINTR). Stack entries are made via the stack quadruple (STK, STKID, STKOVER, STKPTR). The basic FLOP -- construct is implemented with the use of the stack triple (FLOP, CODELOC, FLOPOVER); and the FLOP(ADDR--) construct by the use of the table triple (LABELS, LABELADD, LABELOVER).

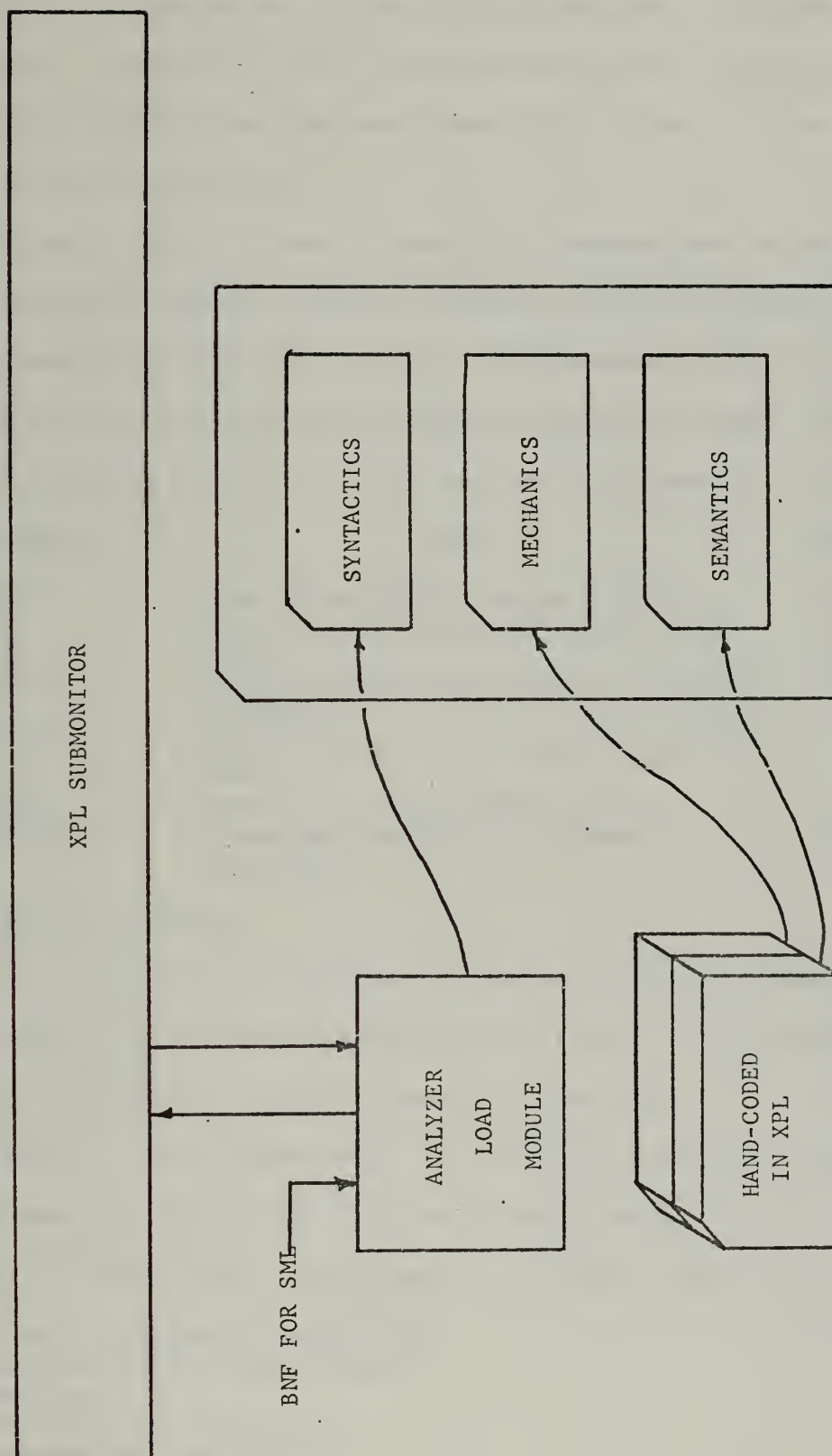


Figure 12

Three tagged locations in Superskeleton are reserved for the insertion of the punched card syntactics, declarations, and semantics respectively. Assuming the user language does not have any primitive variables other than those discussed under SML, the user compiler should then be ready for production.

There are a number of control toggles in Superskeleton to aid the user in debugging both the compiler semantics and user programs. They are complemented by the appearance of a '\$' immediately followed by the appropriate keyword, all contained within a comment statement. The available toggles and their initial state are as follows:

<u>keyword</u>	<u>use</u>	<u>state</u>
LIST	list superskeleton on the printer	on
PRT	trace the allocation of Balgol-2 prt cells	off
CODE	output the generated Balgol-2 code by syllable	off
WORD	output the generated Balgol-2 code by word	off
DUMP	print out the run statistics	off
STUFF	trace the generation of branching operations	off

G. THE BALGOL-2 MACHINE

The Balgol-2 Machine was written for use by students of compiler-writing courses at Naval Postgraduate School [Ref. 7]. The resultant machine is a simple but extremely powerful interpreter.

The Balgol-2 machine was written in the XPL language and essentially is a software simulation of a fictitious stack-oriented computer.

Requiring only about 60K of main memory, it contains mechanisms for:

- evaluating arithmetic expressions
- evaluating boolean expressions
- branching
- subroutines and functions
- execution stack manipulation
- subscripting
- input/output
- dynamic memory allocation

V. CONCLUSIONS

A. THE CAPABILITY OF XPL CGP

As decided at the outset, the primary effort of this thesis would be directed at automating the semantic processing phase of the XPL system. Although the final product, XPL CGP, was developed without any particular user language in mind, the final test was naturally the implementation of a user compiler. Accordingly, Sample Language (BNF is given in Appendix I) was invented. The input to Prescanner, given in Appendix J, was encoded in less than three hours (including the BNF). One debugging run was required and the user compiler was implemented with the XPL CGP output shown in Appendix K. A number of Sample Language programs were subsequently compiled by the user compiler and successfully executed using the Balgol-2 machine.

The total time required for the complete implementation of Sample Language was approximately three hours of user time and four minutes of CPU time. In brief, the performance of XPL CGP was impressive when compared to the time and effort that would have been required using the original XPL system.

Sample Language does not, of course, test all of the capabilities of SML or XPL CGP. Nor is the time that was necessary to implement Sample Language necessarily representative. More complex languages will naturally require more time to encode and debug. Additionally, the constructs available in SML may not be sufficient to describe some functions of these languages.

B. SUGGESTIONS FOR FURTHER RESEARCH

Particular problems which have come to the fore and some thoughts about extending the capability of SML suggest a number of possible improvements to both SML and XPL CGP.

1. The BNF for SML

SML was, of course, developed piecemeal. Some of the resulting keywords are thus either unwieldy or not suggestive enough of their function. It would be desirable to re-examine the BNF and relabel the constructs accordingly. For example; CODE(OP---) and FLOP(ADDR---) would probably be better labelled as OPERATOR and SAVE-ADDRESS respectively. In addition to simply changing the BNF, this relabelling would necessitate checking the code generators to ensure that the correct production stack entry is selected.

2. Reserved Word Problem

All of the declared variables in Superskeleton, and there are a large number of them, are reserved words in the XPL CGP. To minimize the possibility of conflicts with user identifiers, it is suggested that the Superskeleton identifiers all be changed to begin with an uncommon alphabet element. The user could then freely use identifiers beginning with any other element.

Such a change would, of course, necessitate changing each identifier for every occurrence in Superskeleton. The best method would probably be a short XPL routine with Superskeleton as input and punched card output.

3. Indefinite Tables and Stacks

Indefinite tables and stacks (those declared with a '\$' number part) are currently implemented with a very inefficient list structure.

It seems certain that they could be better implemented with some form of mapping function which would allocate, map, and deallocate appropriate n-tuples of words.

In addition to the development of an efficient mapping function, such an improvement would probably require some modifications to both Semantic Preprocessor and Superskeleton.

4. Extensions to SML

There are a number of constructs which might be added to SML to further automate the implementation of some user language structures.

a. Automatic Blocking

Some form of automatic blocking would probably be useful for such language structures as nested BEGIN-END blocks or PROCEDURES. One possible approach would be to implement a BLOCK command which would stack the state of all other significant data structures and possibly perform some sort of re-initialization. An UNBLOCK command could be used to return the program to its previous state.

b. Parameters

The addition of a formal mechanism to pass procedure and function parameters would probably be highly useful. One method of implementation might be to invoke a PARAMETER(<list operand>) command which would dynamically allocate a chained list of the parameters, stack the list pointer for recursive calls, and pass the pointer address to the called procedure or function. A RETURN command would pop the stack and make a branch to the appropriate location.

5. Extensions to XPL CGP

It is not entirely impossible that XPL CGP can be extended to form a true compiler-compiler, limited only by the parsing algorithm.

Such an extension is envisioned as developing a Mechanical Meta-Language and a Mechanical Preprocessor.

The Mechanical Meta-Language would contain constructs which would enable the user to define the desired machine environment in terms of machine registers, instruction format, format interpretation, and an instruction set. The Mechanical Preprocessor would probably produce either a set of tables or a set of declarations, or both. This output would then be inserted into Superskeleton to complete the user compiler.

Such an extension would most certainly require some procedure changes in Superskeleton, particularly for code emission procedures. It also would possibly require some change to SML, notably in the CODE commands.

C. CONCLUDING REMARKS

It has been shown by implementation that the XPL compiler-generator system can be extended to include an automatic semantic processing capability. The extended system, XPL CGP, has been tested and its performance was impressive. Suggestions have been made for improvements in the semantic meta-language base of XPL CGP to further enhance its capability. Finally, it has been postulated that XPL CGP can be extended one step further by automating the mechanics of the system. As there appear to be no insurmountable barriers to this final extension, it is concluded that the XPL system can indeed be developed into a compiler-compiler.

SKELETON

[illegible]

/* SKELETON THE PROTO-COMPILER OF THE XPL SYSTEM

W. M. MCKEEMAN	J. J. HORNING	D. B. WORTMAN
INFORMATION & COMPUTER SCIENCE,	COMPUTER SCIENCE DEPARTMENT,	COMPUTER SCIENCE DEPARTMENT,
UNIVERSITY OF CALIFORNIA AT	STANFORD UNIVERSITY,	STANFORD UNIVERSITY,
SANTA CRUZ, CALIFORNIA 95060	STANFORD, CALIFORNIA 94305	STANFORD, CALIFORNIA 94305
DEVELOPED AT THE STANFORD COMPUTATION CENTER, CAMPUS FACILITY, 1966-69 AND THE UNIVERSITY OF CALIFORNIA COMPUTATION CENTER, SANTA CRUZ, 1968-69.		

DISTRIBUTED THROUGH THE SHARE ORGANIZATION.
THIS VERSION OF SKELETON IS A SYNTAX CHECKER FOR THE FOLLOWING GRAMMAR:

```

<PROGRAM> ::= <STATEMENT LIST>

<STATEMENT LIST> ::= <STATEMENT>
                    | <STATEMENT LIST> <STATEMENT>

<STATEMENT> ::= <ASSIGNMENT> ;

<ASSIGNMENT> ::= <VARIABLE> = <EXPRESSION>

```



```

<EXPRESSION> ::= <ARITH EXPRESSION>
                | <IF CLAUSE> THEN <EXPRESSION> ELSE <EXPRESSION>

<IF CLAUSE> ::= IF <LOG EXPRESSION>

<LOG EXPRESSION> ::= TRUE
                  | FALSE
                  | <EXPRESSION> <RELATION> <EXPRESSION>
                  | <IF CLAUSE> THEN <LOG EXPRESSION> ELSE <LOG EXPRESSION>

<RELATION> ::= =
             | <
             | >

<ARITH EXPRESSION> ::= <TERM>
                    | <ARITH EXPRESSION> + <TERM>
                    | <ARITH EXPRESSION> - <TERM>

<TERM> ::= <PRIMARY>
         | <TERM> * <PRIMARY>
         | <TERM> / <PRIMARY>

<PRIMARY> ::= <VARIABLE>
            | <NUMBER>
            | ( <EXPRESSION> )

<VARIABLE> ::= <IDENTIFIER>
            | <VARIABLE> ( <EXPRESSION> )

```

*/

```

/* FIRST WE INITIALIZE THE GLOBAL CONSTANTS THAT DEPEND UPON THE INPUT
   GRAMMAR. THE FOLLOWING CARDS ARE PUNCHED BY THE SYNTAX PRE-PROCESSOR */

```

```

DECLARE NSY LITERALLY '32', NT LITERALLY '18';
DECLARE V(NSY) CHARACTER(1) INITIAL (',', <ERROR: TOKEN = 0>, '=', '<', '>',
',', '<NUMBER>', '<IDENTIFIER>', '<TERM>', '<PROGRAM>', '<PRIMARY>',
',', '<VARIABLE>', '<RELATION>', '<STATEMENT>', '<IF CLAUSE>', '<ASSIGNMENT>',
',', '<EXPRESSION>', '<STATEMENT LIST>', '<ARITH EXPRESSION>',
',', '<LOG EXPRESSION>', '<ELSE>', '<ELSE>');
DECLARE V_INDEX(12) BIT(16) INITIAL ( 1, 11, 12, 13, 16, 17, 17, 18, 18,
18, 18, 19);
DECLARE C1(NSY) BIT(38) INITIAL (
"(2) 00000 00000 00000 00000",
"(2) 00000 00000 00200 0002",
"(2) 00000 00003 03000 0033",
"(2) 00000 00002 02000 0022");

```



```

DECLARE TRIPLE_INDEX(14) BIT(8) INITIAL ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1);
DECLARE PR_INDEX(32) BIT(8) INITIAL ( 1, 2, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 7, 7, 7,
7, 9, 10, 11, 12, 13, 16, 16, 19, 20, 20, 22, 22, 22, 22, 25, 26, 27, 29, 29,
29);

/* END OF CARDS PUNCHED BY SYNTAX */
/* DECLARATIONS FOR THE SCANNER */

/* TOKEN IS THE INDEX INTO THE VOCABULARY V() OF THE LAST SYMBOL SCANNED,
CP IS THE POINTER TO THE LAST CHARACTER SCANNED IN THE CARDIMAGE,
BCD IS THE LAST SYMBOL SCANNED (LITERAL CHARACTER STRING). */
DECLARE (TOKEN, CP) FIXED, BCD CHARACTER;

/* SET UP SOME CONVENIENT ABBREVIATIONS FOR PRINTER CONTROL */
DECLARE EJECT_PAGE LITERALLY, OUTPUT(1) = PAGE;
PAGE_CHARACTER INITIAL (.1.), DOUBLE_CHARACTER INITIAL ('0'),
DOUBLE_SPACE LITERALLY, OUTPUT(1) = DOUBLE;
X70_CHARACTER INITIAL ('');

/* LENGTH OF LONGEST SYMBOL IN V */
DECLARE (RESERVED_LIMIT, MARGIN_CHOP) FIXED;

/* CHARTYPE() IS USED TO DISTINGUISH CLASSES OF SYMBOLS IN THE SCANNER.
TX() IS A TABLE USED FOR TRANSLATING FROM ONE CHARACTER SET TO ANOTHER.
CONTROL() HOLDS THE VALUE OF THE COMPILER CONTROL TOGGLES SET IN $ CARDS.
NOT_LETTER_OR_DIGIT() IS SIMILIAR TO CHARTYPE() BUT USED IN SCANNING
IDENTIFIERS ONLY.

ALL ARE USED BY THE SCANNER AND CONTROL() IS SET THERE.

DECLARE (CHARTYPE, TX) (255) BIT(8),
        (CONTROL, NOT_LETTER_OR_DIGIT)(255) BIT(1);

/* ALPHABET CONSISTS OF THE SYMBOLS CONSIDERED ALPHABETIC IN BUILDING
IDENTIFIERS */
DECLARE ALPHABET_CHARACTER INITIAL ('ABCDEFGHIJKLMNOPQRSTUVWXYZ_$@#');

/* BUFFER HOLDS THE LATEST CARDIMAGE,
TEXT HOLDS THE PRESENT STATE OF THE INPUT TEXT
(NOT INCLUDING THE PORTIONS DELETED BY THE SCANNER),
TEXT_LIMIT IS A CONVENIENT PLACE TO STORE THE POINTER TO THE END OF TEXT,
CARD_COUNT IS INCREMENTED BY ONE FOR EVERY SOURCE CARD READ,
ERROR_COUNT TABULATES THE ERRORS AS THEY ARE DETECTED,
SEVERE_ERRORS TABULATES THOSE ERRORS OF FATAL SIGNIFICANCE.
*/

```



```

DECLARE (BUFFER, TEXT) CHARACTER,
(TEXT_LIMIT, CARD_COUNT, ERROR_COUNT, SEVERE_ERRORS, PREVIOUS_ERROR) FIXED
;

/* NUMBER_VALUE CONTAINS THE NUMERIC VALUE OF THE LAST CONSTANT SCANNED,
**/
DECLARE NUMBER_VALUE FIXED;

/* EACH OF THE FOLLOWING CONTAINS THE INDEX INTO V() OF THE CORRESPONDING
SYMBOL. WE ASK: IF TOKEN = IDENT ETC. */
DECLARE (IDENT, NUMBER, DIVIDE, EOFILE) FIXED;

/* STOPIT() IS A TABLE OF SYMBOLS WHICH ARE ALLOWED TO TERMINATE THE ERROR
FLUSH PROCESS. IN GENERAL THEY ARE SYMBOLS OF SUFFICIENT SYNTACTIC
HIERARCHY THAT WE EXPECT TO AVOID ATTEMPTING TO START CHECKING AGAIN
RIGHT INTO ANOTHER ERROR PRODUCING SITUATION. THE TOKEN STACK IS ALSO
FLUSHED DOWN TO SOMETHING ACCEPTABLE TO A STOPIT() SYMBOL.
FAILSOFT IS A BIT WHICH ALLOWS THE COMPILER ONE ATTEMPT AT A GENTLE
RECOVERY. THEN IT TAKES A STRONG HAND. WHEN THERE IS REAL TROUBLE
COMPILING IS SET TO FALSE, THEREBY TERMINATING THE COMPILATION.
**/
DECLARE STOPIT(100) BIT(1), (FAILSOFT, COMPILING) BIT(1);

DECLARE S CHARACTER; /* A TEMPORARY USED VARIOUS PLACES */

/* THE ENTRIES IN PRMASK() ARE USED TO SELECT OUT PORTIONS OF CODED
PRODUCTIONS AND THE STACK TOP FOR COMPARISON IN THE ANALYSIS ALGORITHM */
DECLARE PRMASK(5) FIXED INITIAL (0, 0, "FF", "FFFF", "FFFFFFF", "FFFFFFF");

/* THE PROPER SUBSTRING OF POINTER IS USED TO PLACE AN UNDER THE POINT
OF DETECTION OF AN ERROR DURING CHECKING. IT MARKS THE LAST CHARACTER
SCANNED. */
DECLARE POINTER CHARACTER INITIAL (' ');

DECLARE CALLCOUNT(20) FIXED /* COUNT THE CALLS OF IMPORTANT PROCEDURES */
INITIAL(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

/* RECORD THE TIMES OF IMPORTANT POINTS DURING CHECKING */
DECLARE CLOCK(5) FIXED;

/* COMMONLY USED STRINGS */
DECLARE X1 CHARACTER INITIAL(' '); X4 CHARACTER INITIAL(' ');
DECLARE PERIOD CHARACTER INITIAL ('. ');

/* TEMPORARIES USED THROUGHOUT THE COMPILER */
DECLARE (I, J, K, L) FIXED;

```



```

DECLARE TRUE LITERALLY '1', FALSE LITERALLY '0', FOREVER LITERALLY 'WHILE 1';

/* THE STACKS DECLARED BELOW ARE USED TO DRIVE THE SYNTACTIC
   ANALYSIS ALGORITHM AND STORE INFORMATION RELEVANT TO THE INTERPRETATION
   OF THE TEXT. THE STACKS ARE ALL POINTED TO BY THE STACK POINTER SP. */

DECLARE STACKSIZE LITERALLY '75', /* SIZE OF STACK */
DECLARE PARSE_STACK (STACKSIZE) BIT(8); /* TOKENS OF THE PARTIALLY PARSED
                                           TEXT */
DECLARE VAR (STACKSIZE) CHARACTER; /* EBCDIC NAME OF ITEM */
DECLARE FIXV (STACKSIZE) FIXED; /* FIXED (NUMERIC) VALUE */

/* SP POINTS TO THE RIGHT END OF THE REDUCIBLE STRING IN THE PARSE STACK,
   MP POINTS TO THE LEFT END, AND
   MPPI = MP+1.
*/
DECLARE (SP, MP, MPPI) FIXED;

/*
PROCEDURES

PAD: PROCEDURE (STRING, WIDTH) CHARACTER;
      DECLARE STRING CHARACTER, (WIDTH, L) FIXED;

      L = LENGTH(STRING);
      IF L >= WIDTH THEN RETURN STRING;
      ELSE RETURN STRING || SUBSTR(X70, 0, WIDTH-L);
      END PAD;

I_FORMAT:
PROCEDURE (NUMBER, WIDTH) CHARACTER;
      DECLARE (NUMBER, WIDTH, L) FIXED;
      STRING = NUMBER;
      L = LENGTH(STRING);
      IF L >= WIDTH THEN RETURN STRING;
      ELSE RETURN SUBSTR(X70, 0, WIDTH-L) || STRING;
      END I_FORMAT;

ERROR: PROCEDURE (MSG, SEVERITY);

```



```

IF CONTROL(BYTE('M')) THEN OUTPUT = BUFFER;
ELSE IF CONTROL(BYTE('L')) THEN
  OUTPUT = I_FORMAT (CARD_COUNT, 4) || ' ' || ' ' || BUFFER || ' ' || REST;
CP = 0;
END GET_CARD;

```

```

/*
THE SCANNER PROCEDURES
*/

```

```

CHAR: PROCEDURE;
/* USED FOR STRINGS TO AVOID CARD BOUNDARY PROBLEMS */
CP = CP + 1;
IF CP <= TEXT_LIMIT THEN RETURN;
CALL GET_CARD;
END CHAR;

```

```

SCAN: PROCEDURE;
DECLARE (S1, S2) FIXED;
CALLCOUNT(3) = CALLCOUNT(3) + 1;
FAILSOFT = TRUE;
BCD = ' '; NUMBER_VALUE = 0;
SCAN1:
DO
  DC
  FOREVER;
  IF CP > TEXT_LIMIT THEN CALL GET_CARD;
  ELSE
    DO; /* DISCARD LAST SCANNED VALUE */
      TEXT_LIMIT = TEXT_LIMIT - CP;
      TEXT = SUBSTR(TEXT, CP);
      CP = 0;
    END;
    /* BRANCH ON NEXT CHARACTER IN TEXT
    DO CASE CHARTYPE(BYTE(TEXT));

```

```

/* CASE 0 */
/* ILLEGAL CHARACTERS FALL HERE */
CALL ERROR ('ILLEGAL CHARACTER: ' || SUBSTR(TEXT, 0, 1));
/* CASE 1 */
/* BLANK */
DO;
  CP = 1;
  DO WHILE BYTE(TEXT, CP) = BYTE(' ') & CP <= TEXT_LIMIT;

```



```

CP = CP + 1;
END;
CP = CP - 1;
END;

/* CASE 2 */
/* NOT USED IN SKELETON (BUT USED IN XCOM) */
;

/* CASE 3 */
/* NOT USED IN SKELETON (BUT USED IN XCOM) */
;

/* CASE 4 */
DO FOREVER; /* A LETTER: IDENTIFIERS AND RESERVED WORDS */
DO CP = CP + 1 TO TEXT_LIMIT;
IF NOT LETTER OR DIGIT(BYTE(TEXT, CP)) THEN
DO; /* END OF IDENTIFIER */
IF CP > 0 THEN BCD = BCD || SUBSTR(TEXT, 0, CP);
S1 = LENGTH(BCD);
IF S1 > 1 THEN IF S1 <= RESERVED_LIMIT THEN
/* CHECK FOR RESERVED_WORDS */
DO I = V_INDEX(S1-1) TO V_INDEX(S1) - 1;
IF BCD = V(I) THEN
DO; TOKEN = I;
RETURN;
END;
END; /* RESERVED_WORDS EXIT HIGHER: THEREFORE <IDENTIFIER> */
TOKEN = IDENT;
RETURN;
END;

END; /* END OF CARD */
BCD = BCD || TEXT;
CALL GET_CARD;
CP = -1;
END;

/* CASE 5 */
DO; /* DIGIT: A NUMBER */
TOKEN = NUMBER;
DO FOREVER;
DO CP = CP TO TEXT_LIMIT;
S1 = BYTE(TEXT, CP);

```



```

IF S1 < "F0" THEN RETURN;
NUMBER_VALUE = 10*NUMBER_VALUE + S1 - "F0";
END;
CALL GET_CARD;
END;

/* CASE 6 */
DO; /* A /: MAY BE DIVIDE OR START OF COMMENT */
CALL CHAR;
IF BYTE(TEXT, CP) /= BYTE(' ') THEN
DO;
TOKEN = DIVIDE;
RETURN;
END;
/* WE HAVE A COMMENT */
S1, S2 = BYTE(' ');
DO WHILE S1 /= BYTE(' ');
IF S1 = BYTE('$') THEN
DO; /* A CONTROL CHARACTER */
CONTROL(S2) = CONTROL(S2);
IF S2 = BYTE('T') THEN CALL UNTRACE;
ELSE IF S2 = BYTE('U') THEN CALL UNTRACE;
ELSE IF S2 = BYTE('I') THEN
IF CONTROL(S2) THEN
MARGIN_CHOP = TEXT_LIMIT - CP + 1;
ELSE
MARGIN_CHOP = 0;
END;
S1 = S2;
CALL CHAR;
S2 = BYTE(TEXT, CP);
END;

/* CASE 7 */ /* SPECIAL CHARACTERS */
DO; TOKEN = TX(BYTE(TEXT));
CP = 1;
RETURN;
END;

/* CASE 8 */ /* NOT USED IN SKELETON (BUT USED IN XCOM) */
END; /* OF CASE ON CHARTYPE */
CP = CP + 1; /* ADVANCE SCANNER AND RESUME SEARCH FOR TOKEN */

```



```
END;
END SCAN;
```

```
/*
```

```
TIME AND DATE
```

```
*/
```

```
PRINT TIME:
PROCEDURE (MESSAGE, CHARACTER, T, FIXED);
DECLARE MESSAGE = MESSAGE || T/360000 || ':' || T MOD 360000 / 6000 || ':' ||
    T MOD 6000 / 100 || '.';
T = T MOD 100; /* DECIMAL FRACTION */
IF T < 10 THEN MESSAGE = MESSAGE || '0';
OUTPUT = MESSAGE || T || '.';
END PRINT_TIME;
```

```
PRINT DATE AND TIME:
PROCEDURE (MESSAGE, CHARACTER, D, T);
DECLARE MESSAGE, CHARACTER, (D, T, YEAR, DAY, M) FIXED;
DECLARE MONTH(11) CHARACTER, INITIAL ('JANUARY', 'FEBRUARY', 'MARCH',
    'APRIL', 'MAY', 'JUNE', 'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER',
    'NOVEMBER', 'DECEMBER');
DAYS(12) FIXED INITIAL (0, 31, 60, 91, 121, 152, 182, 213, 244, 274,
    305, 335, 366);
YEAR = D/1000 + 1900;
DAY = D MOD 1000;
IF (YEAR & "3") /= 0 THEN IF DAY > 59 THEN DAY = DAY + 1; /* ~ LEAP YEAR */
M = 1;
DO WHILE DAY > DAYS(M): M = M + 1; END;
CALL PRINT_TIME(MESSAGE || MONTH(M-1) || X1 || DAY-DAYS(M-1) || ' ',
    || YEAR || '.' || CLOCK TIME = ' ', T);
END PRINT_DATE_AND_TIME;
```

```
/*
```

```
INITIALIZATION
```

```
*/
```

```
INITIALIZATION:
```

```
PROCEDURE;
EJECT PAGE;
CALL PRINT_DATE_AND_TIME (' SYNTAX CHECK --- STANFORD UNIVERSITY --- SKELETON
    III VERSION OF ', DATE_OF_GENERATION, TIME_OF_GENERATION);
DOUBLE SPACE;
CALL PRINT_DATE_AND_TIME ('TODAY IS ', DATE, TIME);
DOUBLE SPACE;
```



```

DO I = 1 TO NT;
  S = V(I);
  IF S = <NUMBER>, THEN NUMBER = I; ELSE
  IF S = <IDENTIFIER>, THEN IDENT = I; ELSE
  IF S = </>, THEN DIVIDE = I; ELSE
  IF S = <!-->, THEN EOFILE = I; ELSE
  IF S = <:-->, THEN STOPIT(I) = TRUE; ELSE
  ;
END;
IF IDENT = NT THEN RESERVED_LIMIT = LENGTH(V(NT-1));
ELSE RESERVED_LIMIT = LENGTH(V(NT));
V(EOFILE) = 'EOF';
STOPIT(EOFILE) = TRUE;
CHARTYPE(BYTE(' ')) = 1;
DO I = 0 TO 255;
  NOT_LETTER_OR_DIGIT(I) = TRUE;
END;
DO I = 0 TO LENGTH(ALPHABET) - 1;
  TX(J) = I;
  NOT_LETTER_OR_DIGIT(J) = FALSE;
  CHARTYPE(J) = 4;
END;
DO I = 0 TO 9;
  J = BYTE('0123456789', I);
  NOT_LETTER_OR_DIGIT(J) = FALSE;
  CHARTYPE(J) = 5;
END;
DO I = V_INDEX(0) TO V_INDEX(1) - 1;
  J = BYTE(V(I));
  TX(J) = I;
  CHARTYPE(J) = 7;
END;
CHARTYPE(BYTE('/')) = 6;
/* FIRST SET UP GLOBAL VARIABLES CONTROLLING SCAN, THEN CALL IT */
CP = 0; TEXT_LIMIT = -1;
TEXT = '';
CONTROL(BYTE('L')) = TRUE;
CALL SCAN;

/* INITIALIZE THE PARSE STACK */
SP = 1; PARSE_STACK(SP) = EOFILE;

END INITIALIZATION;

```



```

DUMPIT:
PROCEDURE; /* DUMP OUT THE STATISTICS COLLECTED DURING THIS RUN */
DOUBLE SPACE;
/* PUT OUT THE ENTRY COUNT FOR IMPORTANT PROCEDURES */

OUTPUT = 'STACKING DECISIONS= ' || CALLCOUNT(1);
OUTPUT = 'SCAN' = ' || CALLCOUNT(3);
OUTPUT = 'FREE STRING AREA = ' || FREELIMIT - FREEBASE;
END DUMPIT;

STACK_DUMP:
PROCEDURE;
DECLARE LINE CHARACTER;
LINE = 'PARTIAL PARSE TO THIS POINT IS: ';
DO I = 2 TO SP;
IF LENGTH(LINE) > 105 THEN
DO; OUTPUT = LINE;
LINE = X4;
END;
LINE = LINE || X1 || V(PARSE_STACK(I));
END;
OUTPUT = LINE;
END STACK_DUMP;

/* THE SYNTHESIS ALGORITHM FOR XPL */

SYNTHESIZE:
PROCEDURE(PRODUCTION_NUMBER);
DECLARE PRODUCTION_NUMBER FIXED;

/* THIS PROCEDURE IS RESPONSIBLE FOR THE SEMANTICS (CODE SYNTHESIS), IF
ANY OF THE SKELETON COMPILER. ITS ARGUMENT IS THE NUMBER OF THE
PRODUCTION WHICH WILL BE APPLIED IN THE PENDING REDUCTION. THE GLOBAL
VARIABLES MP AND SP POINT TO THE BOUNDS IN THE STACKS OF THE RIGHT PART
OF THIS PRODUCTION.
NORMALLY, THIS PROCEDURE WILL TAKE THE FORM OF A GIANT CASE STATEMENT
ON PRODUCTION_NUMBER. HOWEVER, THE SYNTAX CHECKER HAS SEMANTICS (THE
TERMINATION OF CHECKING) ONLY FOR PRODUCTION 1. */

IF PRODUCTION_NUMBER = 1 THEN

/* <PROGRAM> ::= <STATEMENT LIST> */

```



```

DO; IF MP ^= 2 THEN /* WE DIDN'T GET HERE LEGITIMATELY */
DO; CALL ERROR ('EOF AT INVALID POINT', 1);
CALL STACK_DUMP;
END;
COMPILING = FALSE;
END;
END SYNTHESIZE;

/*          SYNTACTIC PARSING FUNCTIONS          */

RIGHT_CONFLICT:
PROCEDURE (LEFT) BIT(1);
DECLARE LEFT FIXED;
/* THIS PROCEDURE IS TRUE IF TOKEN IS A LEGAL RIGHT CONTEXT OF LEFT */
RETURN ("CQ" & SHL(BYTE(C1(LEFT)), SHR(TOKEN,2)), SHL(TOKEN,1)
& "C6")) = 0;
END RIGHT_CONFLICT;

RECOVER:
PROCEDURE;
/* IF THIS IS THE SECOND SUCCESSIVE CALL TO RECOVER, DISCARD ONE SYMBOL */
IF ~ FAILSOFT THEN CALL SCAN;
IF ~ FAILSOFT = FALSE;
DO WHILE ~ STOPIT(TOKEN);
CALL SCAN; /* TO FIND SOMETHING SOLID IN THE TEXT */
END;
DC WHILE RIGHT_CONFLICT (PARSE_STACK(SP));
IF SP > 2 THEN SP = SP - 1; /* AND IN THE STACK */
ELSE CALL SCAN; /* BUT DON'T GO TOO FAR */
END;
OUTPUT = 'RESUME: ' || SUBSTR(POINTER, TEXT_LIMIT-CP+MARGIN_CHOP+7);
END RECOVER;

STACKING:
PROCEDURE BIT(1); /* STACKING DECISION FUNCTION */
CALLCOUNT(1) = CALLCOUNT(1) + 1;
DO FOREVER; /* UNTIL RETURN */
DO CASE SHR(BYTE(C1(PARSE_STACK(SP)), SHR(TOKEN,2)), SHL(3-TOKEN,1)&6)&3;
/* CASE 0 */
DO; /* ILLEGAL SYMBOL PAIR */

```



```

CALL ERROR('ILLEGAL SYMBOL PAIR: ' || V(PARSE_STACK(SP)) || X1 ||
V(TOKEN), 1);
CALL STACK_DUMP;
CALL RECOVER;
END;

/* CASE 1 */
RETURN TRUE; /* STACK TOKEN */

/* CASE 2 */
RETURN FALSE; /* DON'T STACK IT YET */

/* CASE 3 */
DO; /* MUST CHECK TRIPLES */
J = SHL(PARSE_STACK(SP-1), 16) ÷ SHL(PARSE_STACK(SP), 8) + TOKEN;
I = -1; K = NCITRIPLES + 1; /* BINARY SEARCH OF TRIPLES */
DO WHILE I + 1 < K;
L = SHR(I+K, 1);
IF CITRIPLES(L) > J THEN K = L;
ELSE IF CITRIPLES(L) < J THEN I = L;
ELSE RETURN TRUE; /* IT IS A VALID TRIPLE */
END;
RETURN FALSE;
END;

END; /* OF DO CASE */
/* OF DO FOREVER */
END STACKING;

PR_OK:
/* DECISION PROCEDURE FOR CONTEXT CHECK OF EQUAL OR IMBEDDED RIGHT PARTS */
DECLARE (H, I, J, PRD) FIXED;
DO CASE CONTEXT_CASE(PRD);
/* CASE 0 -- NO CHECK REQUIRED */
RETURN TRUE;
/* CASE 1 -- RIGHT CONTEXT CHECK */
RETURN ~ RIGHT_CONFLICT (HDTB(PRD));
/* CASE 2 -- LEFT CONTEXT CHECK */

```



```

DO; H = HDTB(PRD) - NT;
    I = PARSE_STACK(SP - PLENGTH(PRD));
    DO J = LEFT_INDEX(H-1) TO LEFT_INDEX(H) - 1;
        IF LEFT_CONTEXT(J) = I THEN RETURN TRUE;
    END;
    RETURN FALSE;
END;

/* CASE 3 -- CHECK TRIPLES */

DO; H = HDTB(PRD) - NT;
    I = SHL(PARSE_STACK(SP - PLENGTH(PRD)), 8) + TOKEN;
    DO J = TRIPLE_INDEX(H-1) TO TRIPLE_INDEX(H) - 1;
        IF CONTEXT_TRIPLE(J) = I THEN RETURN TRUE;
    END;
    RETURN FALSE;
END;

END; /* OF DO CASE */
END PR_OK;

```

/*

ANALYSIS ALGORITHM

*/

```

REDUCE:
PROCEDURE:
DECLARE (I, J, PRD) FIXED;
/* PACK STACK TOP INTO ONE WORD */
DO I = SP - 4 TO SP - 1;
    J = SHL(J, 8) + PARSE_STACK(I);
END;

DO PRD = PR_INDEX(PARSE_STACK(SP-1)) TO PR_INDEX(PARSE_STACK(SP)) - 1;
IF (PRMASK(PLENGTH(PRD)) & J) = PRTB(PRD) THEN
    IF PR_OK(PRD) THEN
        DO; MP = 7* AN ALLOWED REDUCTION */
            MP = SP - PLENGTH(PRD) + 1; MP1 = MP + 1;
            CALL SYNTHESIZE(PRTB(PRD));
            SP = MP;
            PARSE_STACK(SP) = HDTB(PRD);
            RETURN;
        END;
    END;

```



```

/* LOOK UP HAS FAILED, ERROR CONDITION */
CALL ERROR('NO PRODUCTION IS APPLICABLE',1);
CALL STACK_DUMP;
FAILSOFT = FALSE;
CALL RECOVER;
END REDUCE;

COMPILE_LOOP:
PROCEDURE;
    COMPILING = TRUE;
    DO WHILE COMPILING;
        DO WHILE STACKING;
            SP = SP + 1;
            IF SP = STACKSIZE THEN
                DO;
                    CALL ERROR ('STACK OVERFLOW *** CHECKING ABORTED ***', 2);
                    RETURN; /* THUS ABORTING CHECKING */
                END;
                PARSE_STACK(SP) = TOKEN;
                VAR(SP) = BCD;
                FIXV(SP) = NUMBER_VALUE;
                CALL SCAN;
            END;
        END;
        CALL REDUCE; DO WHILE COMPILING /*
    END;
    END COMPILE_LOOP;

PRINT_SUMMARY:
PROCEDURE;
    DECLARE I FIXED;
    CALL PRINT_DATE_AND_TIME ('END OF CHECKING ', DATE, TIME);
    OUTPUT = CARD COUNT || ' CARDS WERE CHECKED.';
    IF ERROR_COUNT = 0 THEN OUTPUT = 'NO ERRORS WERE DETECTED.';
    ELSE IF ERROR_COUNT > 1 THEN
        OUTPUT = ERROR_COUNT || ' ERRORS (' || SEVERE_ERRORS
        || ' SEVERE ERRORS WERE DETECTED.';
    ELSE IF SEVERE_ERRORS = 1 THEN OUTPUT = 'ONE SEVERE ERROR WAS DETECTED.';
    ELSE OUTPUT = 'ONE ERROR WAS DETECTED.';
    IF PREVIOUS_ERROR > 0 THEN
        OUTPUT = 'THE LAST DETECTED ERROR WAS ON LINE ' || PREVIOUS_ERROR
        || PERIOD;
    IF CONTROL(BYTE('D')) THEN CALL DUMPIT;

```


APPENDIX B

BNF FOR THE XPL LANGUAGE

```

<PROGRAM>
<STATEMENT LIST>
<STATEMENT>
<BASIC STATEMENT>
<IF STATEMENT>
<STEP DEFINITION>
<ITERATION CONTROL>
<WHILE CLAUSE>
<CASE SELECTOR>
<PROCEDURE DEFINITION>
<PROCEDURE HEAD>
<PROCEDURE NAME>
<PARAMTER LIST>
<PARAMTER HEAD>
<ENDING>

<STATEMENT LIST>
<STATEMENT>
<STATEMENT LIST> <STATEMENT>
<BASIC STATEMENT>
<IF STATEMENT>
<ASSIGNMENT> ;
<GROUP> ;
<PROCEDURE DEFINITION> ;
<RETURN STATEMENT> ;
<CALL STATEMENT> ;
<GO TO STATEMENT> ;
<DECLARATION STATEMENT> ;
;
<LABEL DEFINITION> <BASIC STATEMENT>
<IF CLAUSE> <STATEMENT>
<IF CLAUSE> <TRUE PART> <STATEMENT>
<LABEL DEFINITION> <IF STATEMENT>
IF <EXPRESSION> THEN
<BASIC STATEMENT> ELSE
<GROUP HEAD> <ENDING>
DO ;
DO <STEP DEFINITION> ;
DO <WHILE CLAUSE> ;
DO <CASE SELECTOR> ;
<GROUP HEAD> <STATEMENT>
<VARIABLE> <REPLACE> <EXPRESSION> <ITERATION CONTROL>
TO <EXPRESSION> BY <EXPRESSION>
WHILE <EXPRESSION>
CASE <EXPRESSION>
<PROCEDURE HEAD> <STATEMENT LIST> <ENDING>
<PROCEDURE NAME> ;
<PROCEDURE NAME> <PARAMTER LIST> ;
<LABEL DEFINITION> PROCEDURE
<PARAMTER HEAD> <IDENTIFIER> )
( <PARAMTER HEAD> <IDENTIFIER> ,
END <IDENTIFIER>
END <LABEL DEFINITION> <ENDING>

```


<TERM>

<PRIMARY>

<VARIABLE>

<SUBSCRIPT HEAD>

<CONSTANT>

+ <TERM>
- <TERM>
<PRIMARY>
<TERM> * <PRIMARY>
<TERM> / <PRIMARY>
<TERM> MOD <PRIMARY>
<CONSTANT>
<VARIABLE>
(<EXPRESSION>)
<IDENTIFIER>
<SUBSCRIPT HEAD> <EXPRESSION>)
<IDENTIFIER> (
<SUBSCRIPT HEAD> <EXPRESSION> ,
<STRING>
<NUMBER>

APPENDIX C

BNF FOR SML

```

<SEMANTIC PROGRAM> <HEADER> <SEQUENCE> END
<HEADER> <BEGIN> <DECLARATION SET>
<BEGIN>
<DECLARATION SET>
    <DECLARATION> ;
    <DECLARATION SET> <DECLARATION> ;
    <BRACKETED SENTENCE>
    <SEQUENCE> <BRACKETED SENTENCE>
    <BRACKETED SENTENCE> <DELIMITER 1> <SENTENCE> <DELIMITER 2>
    <DELIMITER 1>
    <DELIMITER 2>
    <SENTENCE>
    # @
    <STATEMENT> <STATEMENT>
    <SENTENCE> <UNCONDITIONAL> ;
    <UNCONDITIONAL>
    <CONDITIONAL>
    <LABEL> <STATEMENT>
    <COMMENT>
    <LDEL> <REMARKS> <RDEL>
    / * /
    <IDENTIFIER> :
    <TABLE DECLARATION>
    <STACK DECLARATION>
    <CELL DECLARATION>
    <TAG DECLARATION>
    <STRING DECLARATION>
    <FLAG DECLARATION>
    <TABLE> <TABLE SPECIFIER>
    <TABLE DECLARATION> , <TABLE SPECIFIER>
    TABLE
    <TABLE SPECIFIER>
    <TABLE HEADER> <FIELDS> )
    <IDENTIFIER> <NUMBER PART>
    ( <NUMBER> :
    ( $ :
    <FIELD>
    <FIELDS> , <FIELD>
    <IDENTIFIER> <TYPE SPECIFIER>
    <BIT>
    CHAR
    INTEGER
    BIT
    <BIT>

```



```

BIT <NUMBER>
<<STACK DECLARATION> <STACK> <STABLE SPECIFIER>
<<STACK> <STACK DECLARATION> , <STABLE SPECIFIER>
<<CELL DECLARATION> <CELL> <IDENTIFIER>
<<TAG DECLARATION> <TAG> <IDENTIFIER> , <IDENTIFIER>
<<STRING DECLARATION> <TAG DECLARATION> , <IDENTIFIER>
<<FLAG DECLARATION> <STRING DECLARATION> , <IDENTIFIER>
<<UNCONDITIONAL> <FLAG> <IDENTIFIER>
<<ASSIGNMENT> <FLAG DECLARATION> , <IDENTIFIER>
<<ASSIGNMENT> <OPERATION>
<<TRANSFER> <OPERATION>
<<LEFT PART> <ARITHMETIC>
<<LEFT PART> <BOOLEAN>
<<PRIMARY> :=
<<TABLE OPERATION>
<<STACK OPERATION>
<<CODE OPERATION>
<<AUXILIARY OPERATION>
<<DELETE OPERATION>
<<DELETE OPERATION> <DELETE> <VARIABLE> )
<<DELETE> <IDENTIFIER>
<<DELETE> ( <IDENTIFIER>
= <PRIMARY>
<<ENTRY POINT1> <LIST OPERAND> )
<<ENTRY POINT1> <PRIMARY> ,
<<ENTRY POINT>
<<ENTER> <VARIABLE> :
<<ENTER> ( <IDENTIFIER>
POP <IDENTIFIER>
NOCODE
<<CODE>
<<CODE>
<<INDIRECT>
<<INDIRECT> <CHARACTER STRING> )
<<CODE 1> <CHARACTER STRING> ,
<<CODE 1> <OPERAND PAIR> ,
<<CODE 1> <CONTENTS> )
<<CODE 1>
<<OPERATOR>
<<OPERATOR> , <CHARACTER STRING>
OP <CHARACTER STRING>
CONT <CHARACTER STRING>
<<AUXILIARY OPERATION> TALLY <IDENTIFIER>

```


<ERROR>
 <TRANSFER>
 <CONDITIONAL>

<IF CLAUSE>
 <IF>
 <THEN CLAUSE>
 <THENDO>
 <ELSE CLAUSE>
 <ELSEDO>
 <BOOLEAN>

<OR>
 <BOOLEAN TERM>
 <AND>
 <BOOLEAN FACTOR>
 <NOT>
 <BOOLEAN PRIMARY>

<TEST>
 <RELATION>

<ARITHMETIC>

<+>
 <->
 <TERM>

<*>
 </>
 <FACTOR>

MINUS <IDENTIFIER>
 <ERROR> <PRIMARY>
 STOP
 ERROR
 JUMP <IDENTIFIER>
 <IF CLAUSE> <THEN CLAUSE> <ELSE CLAUSE>
 <IF> <BOOLEAN>
 IF
 <THENDO> <SENTENCE> END ;
 THENDO ;
 <ELSEDO> <SENTENCE> END ;
 ELSEDO ;
 <BOOLEAN TERM>
 <BOOLEAN> <OR> <BOOLEAN TERM>
 OR
 <BOOLEAN FACTOR>
 <BOOLEAN TERM> <AND> <BOOLEAN FACTOR>
 AND
 <BOOLEAN PRIMARY>
 <NOT> <BOOLEAN PRIMARY>
 NOT
 <ARITHMETIC> <RELATION> <ARITHMETIC>
 <LDEL*> <BOOLEAN> <RDEL*>
 <TEST> <PRIMARY>
 TRUE
 FALSE
 TEST
 • LESS.
 • LEQ.
 • EQ.
 • NEQ.
 • GEO.
 • GTR.
 <TERM>
 <+> <TERM>
 <-> <TERM>
 <ARITHMETIC> <+> <TERM>
 <ARITHMETIC> <-> <TERM>
 + -
 <FACTOR>
 <TERM> <*> <FACTOR>
 <TERM> </> <FACTOR>
 * /
 <PRIMARY> <*> <PRIMARY>
 <FACTOR> <*> <PRIMARY>


```

<*>
<PRIMARY>

<OPERAND>

**
<OPERAND>
<NUMBER>
<BIT STRING>
<CHARACTER STRING>
<LDEL*> <ARITHMETIC> <RDEL*>
<STABLE OPERAND>
<FLOATING OPERAND>
<STORAGE OPERAND>
<CODE OPERAND>
( <LIST OPERAND> )
<ENTRY OPERAND> )
<LOC OPERAND> )
TOP <IDENTIFIER>
<SET OPERAND> )
<LOC> <VARIABLE>
LOC ( <IDENTIFIER>
ENTRY ( <VARIABLE>
<SET POINT> <IDENTIFIER>
<SET POINT 1> <PRIMARY>
<SET POINT 1> <VARIABLE> :
<SET> <IDENTIFIER>
SET (
<FLOP> <PRIMARY> )
<FLOP> <PROGRAM LABEL> )
FLOP (
ADDR <PRIMARY>
<IDENTIFIER>
CODELOC
<PRIMARY> ; <PRIMARY>
<LIST OPERAND> ; <PRIMARY>
<
>

```


THE CGP PRESCANNER

[illegible]

PRES CANNER

DECLARATIONS

DECLARATIONS

```

DECLARE LITERALLY '175', CHARACTER, PRODUCTIONS
BNF_BUFFER(BNF) FIXED INITIAL(0), SYNTAX PRODUCTIONS
BNF_CARD_COUNT FIXED INITIAL(0), POINTER TO BNF_BUFFER
BNF_COUNT FIXED INITIAL(0), NO. OF PRODUCTIONS
BUFFER_CHARACTER, INITIAL(0), GLOBAL TEMPORARY
CARD_COUNT FIXED INITIAL(0), NO. OF CARDS SCANNED
CONTROL_COUNT FIXED INITIAL(0), NO. OF CONTROL CARDS
ERROR_COUNT FIXED INITIAL(0), NO. OF ERRORS
FALSE_LITERALLY '0', GLOBAL LITERAL
FIRST_CARD_BIT(1) INITIAL(0), FIRST CARD FLAG
FIRST_PRODUCTION_BIT(1) INITIAL(1), FIRST PRODUCTION FLAG
PRODUCTIONS_BIT(1) INITIAL(0), PRODUCTION PUNCH FLAG
PUNCH_BIT(1) INITIAL(0), SYNTAX TEMPORARY
S_CHARACTER, '325', MAX SML PRODUCTIONS
SML_LITERALLY '325', CHARACTER, SEMANTIC PRODUCTIONS
SML_BUFFER(SML) FIXED INITIAL(0), POINTER TO SML_BUFFER
SML_CARD_COUNT FIXED INITIAL(0), NO. OF SML PRODUCTIONS
SML_COUNT FIXED INITIAL(0), MATCHING LITERAL
TRUE_LITERALLY '1', UNMATCHED_BIT(1) INITIAL(0), COMMONLY USED STRING
UNMATCHED_BIT(1) INITIAL(0), COMMONLY USED STRING
X1_CHARACTER INITIAL(''), COMMONLY USED STRING
X2_CHARACTER INITIAL(''), COMMONLY USED STRING
X7_CHARACTER INITIAL(''), COMMONLY USED STRING
X10_CHARACTER INITIAL(''), COMMONLY USED STRING

```



```

X90 CHARACTER INITIAL(' '); /* COMMONLY USED STRING */
/*
/* MISCELLANEOUS PROCEDURES
/*
FOR MAT:
PROCEDURE(NUMBER,WIDTH) CHARACTER;
DECLARE (NUMBER,WIDTH,L) FIXED,STRING CHARACTER;
STRING = NUMBER;
L=LENGTH(STRING);
IF L>=WIDTH THEN RETURN STRING;
ELSE RETURN SUBSTR(X10,0,WIDTH-L)||STRING;
END FORMAT;

GET CARD:
PROCEDURE BIT(1);
BUFFER=INPUT;
IF LENGTH(BUFFER)=0
THEN DO;
IF PRODUCTIONS
THEN DO;
OUTPUT(1)='0';
OUTPUT=***ERROR ***EOF MISSING ****;
ERROR_COUNT=ERROR_COUNT+1;
RETURN FALSE;
END;
ELSE RETURN FALSE;
END;
ELSE RETURN TRUE;
END;
DO;
CARD_COUNT=CARD_COUNT+1;
OUTPUT=FORMAT(CARD_COUNT,4)||'|'||BUFFER||'|';
END;
END GET_CARD;

CHECK_CARD:
PROCEDURE;
IF BYTE(BUFFER)=BYTE('$')
THEN RETURN 0;
IF BYTE(BUFFER)=BYTE('<')
```



```

/*
/*
/*

```

MISCELLANEOUS PROCEDURES

```

/*
/*
/*

```

```

PRINT_SUMMARY:
PROCEDURE;
DECLARE I FIXED;
OUTPUT(1) = '1';
OUTPUT = '*****';
OUTPUT(1) = '0';
OUTPUT = CARD_COUNT || ' CARDS WERE EXAMINED, OF WHICH:';
OUTPUT = X7 || ' CONTROL_COUNT || ' WERE CONTROL CARDS,';
OUTPUT = X7 || ' BNF_COUNT || ' WERE SYNTAX CARDS, AND';
OUTPUT = X7 || ' SML_COUNT || ' WERE SEMANTIC CARDS.';
IF ERROR_COUNT = 0
THEN DO;
OUTPUT = 'NO ERRORS WERE DETECTED.';
OUTPUT(1) = 0;
OUTPUT = BNF_CARD_COUNT || ' SYNTAX PRODUCTIONS WILL BE PASSED'
|| ' TO THE ANALYZER.';
DO I = 0 TO BNF_CARD_COUNT;
OUTPUT = X10 || BNF_BUFFER(I);
END;
OUTPUT(1) = '0';
OUTPUT = SML_CARD_COUNT || ' SEMANTIC PRODUCTIONS WILL BE PASSED'
|| ' TO THE SML COMPILER.';
DO I = 0 TO SML_CARD_COUNT;
OUTPUT = X10 || SML_BUFFER(I);
END;
END;
ELSE DO;
OUTPUT(1) = '0';
IF ERROR_COUNT = 1
THEN OUTPUT = 'ONE ERROR WAS DETECTED, JOB TERMINATED.';
ELSE OUTPUT = ERROR_COUNT || ' ERRORS WERE DETECTED, JOB TERMINATED.';
END;
END PRINT_SUMMARY;

CLEANUP:
PROCEDURE;
IF PUNCH
THEN CALL SYNTAX( '$PUNCH' || SUBSTR(X80,0,74));
BNF_CARD_COUNT = BNF_CARD_COUNT - 1;
SML_CARD_COUNT = SML_CARD_COUNT - 1;
END CLEANUP;

BUILD_FILE:
PROCEDURE;

```


THE SYNTACTIC PREPROCESSOR

SYNTACTIC PREPROCESSOR

DECLARATIONS

92


```

T CHARACTER, BIT(8),
TAIL(255) BIT(8),
TERMINATOR FIXED,
TEXT(TEXTLIMIT) BIT(8),
THIS TIME FIXED,
TOKEN FIXED,
TOKEN_SAVE(DEPTH) BIT(8),
TP FIXED,
TP_SAVE(DEPTH) BIT(8),
TRIPLE(MAXNTRIP) FIXED,
TROUBLE(MAXTROUBLE) BIT(8),
TROUBLE1(MAXTROUBLE) BIT(8),
TROUBLE2(MAXTROUBLE) BIT(8),
TROUBLE_COUNT FIXED,
TV(MAXNTRIP) BIT(2),
V(255) CHARACTER, INITIAL(2,1),
VALUE(1) FIXED INITIAL(2,1),
WORK("4000") BIT(8),
X12 CHARACTER INITIAL('

```

```

** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** 
** ** ** ** ** ** ** ** ** ** ** 
** ** ** ** ** 
** ** ** ** 

```

```

IS_HEAD:
PROCEDURE(I,J) BIT(1);
DECLARE(I,J) FIXED;
RETURN 1 & SHR(HEAD_TABLE(SHL(I,5)+SHR(J,3)),J & 7);
END IS_HEAD;

```

```

SET HEAD:
PROCEDURE(I,J);
DECLARE(I,J,K,L) FIXED;
CHANGE = TRUE;
K = SHL(I,5) + SHR(J,3);
L = SHL(1,J&7);
HEAD_TABLE(K) = HEAD_TABLE(K) | L;
END SET_HEAD;

```

```

CLEAR HEADS:
PROCEDURE;
DECLARE I FIXED;
DO I = 0 TO "2000";

```



```

HEAD_TABLE(I) = 0;
END;
END CLEAR_HEADS;

GET: PROCEDURE(I,J) BIT(2);
DECLARE (I,J) FIXED;
RETURN 3 & SHR(WORK(SHL(I,6)+SHR(J,2)),SHL(J & 3,1));
END GET;

SET: PROCEDURE(I,J,VAL);
DECLARE (I,J,K,L,VAL) FIXED;
K = SHL(I,6) + SHR(J,2);
L = SHL(VAL & 3,SHL(J & 3,1));
WORK(K) = WORK(K) | L;
END SET;

CLEAR_WORK:
PROCEDURE;
DECLARE I FIXED;
DO I = 0 TO "4000";
    WORK(I) = 0;
END;
END CLEAR_WORK;

PACK: PROCEDURE(B1,B2,B3,B4) FIXED;
DECLARE (B1,B2,B3,B4) BIT(8);
RETURN SHL(B1,24) + SHL(B2,16) + SHL(B3,8) + B4;
END PACK;

ERROR: PROCEDURE(MESSAGE);
DECLARE MESSAGE CHARACTER;
OUTPUT = '*** ERROR, !! MESSAGE;
ERROR_COUNT = ERROR_COUNT + 1;
END ERROR;

ENTER: PROCEDURE(ENV,VAL);
DECLARE (ENV,VAL,I,J,K) FIXED;
NETRY = NETRY + 1;
I = 0; K = NTRIP + 1;
DO WHILE I + 1 < K;
    J = SHR(I+K,1); ENV
    IF TRIPLE(J) > ENV
    THEN K = J; ELSE

```



```

LINE_OUT:
PROCEDURE(NUMBER,LINE);
DECLARE NUMBER FIXED, (N,LINE) CHARACTER;
N = NUMBER;
NUMBER = 6 - LENGTH(N);
OUTPUT = SUBSTR(EMPTY,C,NUMBER) || N || ' ' || LINE;
END LINE_OUT;

BUILD_CARD:
PROCEDURE(ITEM);
DECLARE ITEM CHARACTER;
IF LENGTH(ITEM) + LENGTH(OUTCARD) >= 80
THEN DO;
IF CONTROL(BYTE('P'))
THEN OUTPUT(PUNCH) = OUTCARD;
IF CONTROL(BYTE('O'))
THEN OUTPUT = '--- CARD OUTPUT --- |' || OUTCARD;
OUTCARD = ' ' || ITEM;
END;
ELSE OUTCARD = OUTCARD || ' ' || ITEM;
END BUILD_CARD;

PUNCH_CARD:
PROCEDURE(ITEM);
DECLARE ITEM CHARACTER;
CALL BUILD_CARD(ITEM);
IF CONTROL(BYTE('P'))
THEN OUTPUT(PUNCH) = OUTCARD;
IF CONTROL(BYTE('O'))
THEN OUTPUT = '--- CARD OUTPUT --- |' || OUTCARD;
OUTCARD = ' ' || ITEM;
END PUNCH_CARD;

PRINT_MATRIX:
PROCEDURE(TITLE,SOURCE);
DECLARE TITLE CHARACTER; SOURCE FIXED;
DECLARE (I,J,K,L,M,N,BOT,TOP,MARGIN_SIZE,NUMBER_ACROSS,WIDE) FIXED,
(MARGIN,LINE,WASTE,BAR,PAGES) CHARACTER,
DIGIT(9) CHARACTER INITIAL ('0','1','2','3','4','5','6','7','8','9'),
NUMBER_HIGH_LITERAL('48',
GS_LITERAL('16',
IF SOURCE = 1
THEN WIDE = NT;
ELSE WIDE = NSY;
MARGIN_SIZE = 5;
DO I = 1 TO NSY;

```



```

IF LENGTH(V(I)) >= MARGIN_SIZE
THEN MARGIN_SIZE = LENGTH(V(I)) + 1;
END;
MARGIN = SUBSTR(' ', MARGIN_SIZE);
WASTE = MARGIN;
NUMBER_ACROSS = (122 - MARGIN_SIZE)/(GS + 1)*GS;
DO I = 0 TO 3;
COUNT(I) = 0;
END;
M = 0;
I = (WIDE-1)/NUMBER_ACROSS + 1;
PAGES = ((NSY-1)/NUMBER_HIGH + 1)*I;
DO I = 0 TO (WIDE-1)/NUMBER_ACROSS;
BOT = NUMBER_ACROSS*I + 1;
TOP = NUMBER_ACROSS * (I+1);
IF TOP > WIDE
THEN TOP = WIDE;
BAR = SUBSTR(WASTE, 1) || '+';
DO L = BOT TO TOP;
IF L MOD GS = 0
THEN BAR = BAR || '+';
END;
TOP MOD GS = 0
IF THEN BAR = BAR || '+';
DO J = 0 TO (NSY-1)/NUMBER_HIGH;
EJECT_PAGE;
M = M + 1;
OUTPUT = TITLE || ' : PAGE ' || M || ' OF ' || PAGES;
DOUBLE_SPACE;
L = 1;
DO WHILE L > 0;
LINE = WASTE;
DO N = BOT TO TOP;
IF N < L
THEN LINE = LINE || ' ';
ELSE LINE = LINE || DIGIT(N/L MOD 10);
IF N MOD GS = 0
THEN LINE = LINE || ' ';
END;
OUTPUT = LINE;
L = L / 10;
END;
OUTPUT = BAR;
N = NUMBER_HIGH*(J+1);
IF N > NSY
THEN N = NSY;

```



```

DO K = NUMBER_HIGH * J + 1 TO N;
  L = LENGTH(V(K));
  LINE = V(K) || SUBSTR(MARGIN,L) || '||';
  DO L = BOT TO TOP;
    IF SOURCE ^= 0
      THEN DO;
        N = GET (K,L);
        LINE = LINE || PRINT(N);
        COUNT(N) = COUNT(N) + 1;
      END;
    ELSE LINE = LINE || PRINT(IS_HEAD (K,L));
    IF L MOD GS = 0
      THEN LINE = LINE || '||';
    END;
    IF TOP MOD GS ^= 0
      THEN LINE = LINE || '||';
    CALL LINE_OUT (K,LINE);
    IF K MOD GS = 0
      THEN OUTPUT = BAR;
    END;
    IF K MOD GS ^= 1
      THEN OUTPUT = BAR;
    END;
  END;
DOUBLE_SPACE;
IF SOURCE ^= 0
  THEN DO;
    OUTPUT = 'TABLE ENTRIES SUMMARY: ';
    DO I = 0 TO 3;
      CALL LINE_OUT (COUNT(I),PRINT(I));
    END;
  END;
END; PRINT_MATRIX;

PRINT TRIPLES:
PROCEDURE(TITLE);
DECLARE TITLE CHARACTER(I,J) FIXED;
IF NTRIP = 0
  THEN DO;
    DOUBLE_SPACE;
    OUTPUT = 'NO TRIPLES REQUIRED.';
    COUNT(1) = C;
    RETURN;
  END;
EJECT PAGE;
OUTPUT = TITLE || '||';
DOUBLE_SPACE;
DO I = 1 TO 3;

```



```

PRINT TIME:
PROCEDURE (I,J) FIXED,T CHARACTER;
DECLARE SPACE;
DOUBLE SPACE;
THIS TIME = TIME;
I = I MOD 100;
J = I / 100;
I = TIME USED WAS ' || I || '.';
IF J < 10 THEN T = T || '0.';
OUTPUT = T || J || 'SECONDS.';
I = THIS TIME - FIRST_TIME;
J = I MOD 100;
I = I / 100;
I = TOTAL TIME IS ' || I || '.';
IF J < 10 THEN T = T || '0.';
OUTPUT = T || J || 'SECONDS.';
LAST TIME = THIS_TIME;
END PRINT_TIME;

LOOK_UP:
PROCEDURE (SYMBOL) BIT(8);
DECLARE SYMBOL CHARACTER;
DECLARE J FIXED;
DO J = 1 TO NSY;
IF V(J) = SYMBOL THEN RETURN J;
END;
IF J = 256
THEN DO;
CALL ERROR ('TOO MANY SYMBOLS');
J = 1;
END;
NSY = J;
V(J) = SYMBOL;
RETURN J;
END LOOK_UP;

EXPAND:
PROCEDURE (F11,P);
DECLARE (F11,I,J,P,OLDP) FIXED;
IF P = OLDP
THEN DO;
OLDP = P;
SP = 2;
STACK(SP) = RIGHT_HEAD(P);
J = PRODUCTION(P);

```



```

END;
END; MIT;
IF STOP ^= BLANK
THEN CP = CP + 1;
T = SUBSTR(CARDIMAGE, LP, CP-LP);
RETURN LOOK_UP(T);
END;
END;
T = '';
RETURN 0;
END SCAN;

GET_CARD:
PROCEDURE BIT(1);
CP = 0;
DO WHILE TRUE;
CARDIMAGE = INPUT;
LONG = LENGTH(CARDIMAGE) - 1;
IF LONG < 0
THEN DO;
STACKING = FALSE;
RETURN FALSE;
END;
BYTE(CARDIMAGE) = DOLLAR
THEN DO;
IF SUBSTR(CARDIMAGE, 1, 3) = 'EDG'
THEN RETURN FALSE;
IF CONTROL(BYTE('L'))
THEN OUTPUT = CARDIMAGE;
CONTROL(BYTE(CARDIMAGE, 1)) = ^ CONTROL(BYTE(CARDIMAGE, 1));
END; ELSE
IF CARDIMAGE ^= EMPTY
THEN RETURN TRUE;
END; GET_CARD;

SORT_V:
PROCEDURE;
DO I = 1 TO NSY;
SORT#(I) = SHL(ON_LEFT(I), 16) | SHL(LENGTH(V(I)), 8) | I;
END;
K, L = NSY;
DO WHILE K <= L;
L = 0;
DO I = 2 TO K;
L = I - 1;
IF SORT#(L) > SORT#(I)

```



```

THEN DO; SORT#(L);
      J = SORT#(L); SORT#(I);
      SORT#(L) = SORT#(I);
      T = V(L);
      V(L) = V(I);
      V(I) = T;
      K = L;
    END;
  END;
DO I = 1 TO NSY;
  INDEX(SORT#(I) & "FF") = I;
END;
NT = NSY;
DO WHILE SORT#(NT) > "10000";
  NT = NT - 1;
END;
DO I = 1 TO NPR;
  LEFT_PART(I) = INDEX(LEFT_PART(I));
  J = INDEX(RIGHT_HEAD(I));
  ON RIGHT(J) = TRUE;
  RIGHT_HEAD(I) = J;
  L = PRODUCTION(I);
  DO K = 0 TO 3;
    J = INDEX(SHR(L,24));
    ON RIGHT(J) = TRUE;
    L = SHL(L,8) + J;
  END;
  PRODUCTION(I) = L;
END;
TERMINATOR = INDEX(1); TRUE;
ON RIGHT(TERMINATOR) = TRUE;
END SORT_V;

PRINT DATE:
PROCEDURE (MESSAGE, D);
DECLARE MESSAGE CHARACTER, D FIXED;
DECLARE MONTH(11) CHARACTER INITIAL ('JANUARY', 'FEBRUARY', 'MARCH',
    'APRIL', 'MAY', 'JUNE', 'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER',
    'NOVEMBER', 'DECEMBER'); DAYS(11) FIXED INITIAL(0,31,60,91,
    121,152,182,213,244,274,305,335);
DECLARE (YEAR, DAY, M) FIXED;
YEAR = D/1000 + 1900;
DAY = D MOD 1000;
IF (YEAR & 3) /= 0
THEN IF DAY > 59
THEN DAY = DAY + 1;

```



```

IF SUBSTR(CARDIMAGE,CP) /= SUBSTR(EMPTY,CP)
THEN CALL ERROR('TOO MANY SYMBOLS IN RIGHT PART.
'DISCARD ' || SUBSTR(CARDIMAGE,CP));
PRODUCTION(NPR) = P;
IF CONTROL(BYTE('L'))
THEN DO;
  IF LEFT_PART(NPR) = LEFT_PART(NPR-1)
  THEN DO;
    I = LENGTH(V(LEFT_PART(NPR)));
    CARDIMAGE = SUBSTR(EMPTY,0,I) || ' ' || ' ';
  END; DO;
  ELSE
    OUTPUT = '';
    CARDIMAGE = V(LEFT_PART(NPR)) || ' ' ::= ' ';
  END; BUILD_RIGHT_PART(NPR);
  CALL LINE_OUT (NPR,CARDIMAGE || T);
END;
IF CONTROL(BYTE('P'))
THEN DO;
  P = PRODUCTION(NPR);
  OUTCARD = '/*'; V(LEFT_PART(NPR)) || ' ' ::= ' ||
  V(RIGHT_HEAD(NPR));
  DO K = 1 TO 4;
    IF I SHR(P,SHL(4-K,3)) & "FFF";
    IF I /= 0
    THEN CALL BUILD_CARD (V(I));
  END; CALL PUNCH_CARD (' *');
END;
END; PRINT TIME;
CALL SORT V;
EJECT PAGE;
OUTPUT = TERMINALS;
DOUBLE_SPACE;
IF NSY - NT > NT
THEN L = NSY - NT;
ELSE L = 1 TO L;
DO I = 1 TO L;
  IF I > NT
  THEN CARDIMAGE = HALF_LINE;
  ELSE DO;
    J = 5 - LENGTH(T);
    CARDIMAGE = SUBSTR(SUBSTR(EMPTY,0,J) || T || ' ' || V(I)
    || HALF_LINE,0,66);

```



```

INDEX(I) = 0;
END;
DO I = 1 TO TROUBLE_COUNT;
  DOUBLE SPACE;
  T1 = TROUBLE1(I);
  T2 = TROUBLE2(I);
  DO P = 1 TO BASIC NPR;
    INTERNAL CHANGE = FALSE;
    S1 = RIGHT_HEAD(P);
    M = PRODUCTION(P);
    DO L = 1 TO 4;
      S2 = SHR(M,24);
      IF S2 = 0
        THEN DO;
          IF S1 = T1 & IS_HEAD (S2,T2)
            THEN DO;
              J, INDEX(T1) = INDEX(T1) + 1;
              IF NSY < 255 THEN NSY = NSY + 1;
              ELSE CALL ERROR ('TOO MANY SYMBOLS');
              S = SUBSTR(ADD_ON,J,1);
              V(NSY) = '<';
              IF V(T1) || S || '>';
              IF NPR < 255
                THEN NPR = NPR + 1;
              ELSE CALL ERROR ('TOO MANY PRODUCTIONS. ');
              LEFT_PART(NPR) = NSY;
              RIGHT_HEAD(NPR) = T1;
              PRODUCTION(NPR) = 0;
              CALL OUTPUT_PRODUCTION (NPR);
              CHANGE = TRUE;
              IF INTERNAL
                THEN M = M & "FFFFFFF" | SHL(NSY,8);
              ELSE RIGHT_HEAD(P) = NSY;
            END;
          INTERNAL = TRUE;
          S1 = S2;
        END;
      END;
    END; PRODUCTION(P) = M;
    IF CHANGE
      THEN CALL OUTPUT_PRODUCTION (P);
    END;
  END;
  TROUBLE_COUNT, ERROR_COUNT = 0;
  END IMPROVE_GRAMMAR;

```

```

/* ** ** ** **

```


/// **

COMPUTE_HEADS:


```

ELSE RETURN FALSE;
END; NF11 >= MAXNF11
THEN DO;
  CALL ERROR ('F11 OVERFLOW');
  NF11 = 1;
END;
DO J = 0 TO NF11 - K;
  F11(NF11PI-J) = F11(NF11-J);
END;
NF11 = NF11PI;
F11(K) = NEW_F11;
RETURN TRUE;
END NEVER_BEEN_HERE;

ADD_ITEM:
PROCEDURE (ITEM);
DECLARE ITEM CHARACTER;
IF LENGTH(CARDIMAGE) + LENGTH(ITEM) > 130
THEN DO;
  OUTPUT = CARDIMAGE;
  CARDIMAGE = '';
END;
CARDIMAGE = CARDIMAGE || ' ' || ITEM;
END ADD_ITEM;

PRINT FORM:
PROCEDURE;
CARDIMAGE = 'LEVEL ' || LEVEL || ': ';;
DO I = 1 TO SP;
  CALL ADD_ITEM (V(STACK(I)));
END;
CALL ADD_ITEM(' | ');
DO I = 0 TO TP - 1;
  CALL ADD_ITEM (V(TEXT(TP-I)));
END;
OUTPUT = CARDIMAGE;
END PRINT_FORM;

APPLY PRODUCTION:
PROCEDURE;
LEVEL = LEVEL + 1;
IF LEVEL > MAXLEVEL
THEN IF LEVEL > DEPTH
THEN DO;
  CALL ERROR ('LEVEL OVERFLOW');
  LEVEL = 1;
END;

```



```

ELSE MAXLEVEL = LEVEL;
MP_SAVE(LEVEL) = MP;
MP = SP;
TP_SAVE(LEVEL) = TP;
P_SAVE(LEVEL) = P;
TOKEN_SAVE(LEVEL) = TOKEN;
STACK(SP) = RIGHT_HEAD(P);
J = PRODUCTION(P);
DO WHILE J /= 0;
  K = SHR(J,24);
  J = SHL(J,8);
  IF K /= 0
    THEN IF SP = STACKLIMIT
      THEN CALL ERROR ('STACK OVERFLOW');
    ELSE DO;
      SP = SP + 1;
      STACK(SP) = K;
    END;
  END;
END;
CONTROL(BYTE('T'))
THEN CALL PRINT_FORM;
END APPLY_PRODUCTION;

```

```

DIS APPLY:
PROCEDURE;
TOKEN = TOKEN_SAVE(LEVEL);
P = P_SAVE(LEVEL);
TP = TP_SAVE(LEVEL);
SP = MP;
MP = MP_SAVE(LEVEL);
STACK(SP) = LEFT_PART(P);
LEVEL = LEVEL - 1;
END DIS_APPLY;

```

```

/* * * * * * * * * * *
/* * * * * * * * * * *
/* * * * * * * * * * *

```

```

DO I = 1 TO NSY;
  INDEX(I) = C;
END;
DO I = 1 TO NPR;
  J = LEFT_PART(I);
  IF J /= LEFT_PART(I-1)
    THEN IF INDEX(J) = 0
      THEN INDEX(J) = I;
  ELSE CALL ERROR ('PRODUCTIONS SEPARATED FOR ' || V(J) ||
    ' WILL BE IGNORED.');
```



```

END; PART(NPR+1) = 0;
LEFT_0;
TP = 0;
MP, SP = 1;
TEXT(0) = TERMINATOR;
STACK(0) = GOAL_SYMBOL;
STACK(1) = GOAL_SYMBOL;
NETRY, NF11, LEVEL, MAXLEVEL = 0;
EJECT_PAGE;
OUTPUT = SENTENTIAL FORM PRODUCTION:'';
DOUBLE_SPACE;
IF CCNTRL(BYTE('T')) THEN CALL PRINT_FORM;

PRODUCTION_LOOP:
DO WHILE SP >= MP;
IF STACK(SP) > NT
THEN DO;
NEW = FALSE;
I = TEXT(TP);
DO IF I = 1; TO NT;
IF I IS HEAD(I, TOKEN)
THEN IF NEVER_BEEN_HERE
THEN NEW = TRUE;
END;
IF NEW
THEN DO;
P = INDEX(STACK(SP));
DO WHILE LEFT_PART(P) = STACK(SP);
CALL APPLY_PRODUCTION;
GO TO PRODUCTION_LOOP;
CONTINUE;
P = P + 1;
END;
END;
END;
IF TP = TEXTLIMIT
THEN CALL ERROR('TEXT OVERFLOW');
ELSE TP = TP + 1;
TEXT(TP) = STACK(SP);
SP = SP - 1;
END;
CALL DIS_APPLY;
IF LEVEL >= 0
THEN GO TO CONTINUE;
IF CCNTRL(BYTE('T'))
THEN DOUBLE_SPACE;
OUTPUT = 'FILE HAS ' || NF11 || ' ELEMENTS.';
OUTPUT = 'THE MAXIMUM DEPTH OF RECURSION WAS ' || MAXLEVEL || ' LEVELS.';

```



```

IF M = 0
THEN DO;
M = J;
J = 0;
L = 7;
END;
ELSE L = 6;
TAIL(I) = M & "FF";
M = M & "FFFFFFF0";
DO WHILE M /= 0;
M = SHR(M,24);
IF K /= 0
THEN DO;
J = SHL(J,8) + K;
L = L - 1;
END;
END;
HEAD(I) = J;
SORT#(I) = SHL(TAIL(I),23) + SHL(L,20) + IND1(I) + IND(I);
INDEX(I) = I;
END;
K,L = NPR;
DO WHILE K <= L;
DO I = -1; 2 TO K;
L = I - 1;
IF SORT#(L) > SORT#(I)
THEN DO;
J = SORT#(L);
SORT#(L) = SORT#(I);
SORT#(I) = J;
J = INDEX(L);
INDEX(L) = INDEX(I);
INDEX(I) = J;
K = L;
END;
END;
END;
INDEX(NPR+1) = 0;
END SORT_PRODUCTIONS;

```

```

/** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** **
/** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** **
/** ** ** ** * * * * *
/** ** ** ** * * * * *

```



```

COMPUTE C1:
PROCEDURE:
DECLARE (CX, CTrip, S1, S2, S3, TR, PR) FIXED;
CALL CLEAR WORK;
NETRY, NTRIP, CTrip = 0;
DO CX = 0 TO 2;
DO P = 1 TO NPR;
DO I = IND(P) TO IND1(P);
CALL EXPAND (F11(I), P);
DO J = 2 TO SP;
K = VALUE(J, SP);
S1 = STACK(J-1);
S2 = STACK(J);
L = STACK(J+1);
DO S3 = 1 TO NT;
IF IS HEAD(L, S3)
THEN DO CASE CX;
/* CASE 0 -- ENTER PAIR */
CALL SET (S2, S3, K);
/* CASE 1 -- ENTER TRIPLE */
IF GET (S2, S3) = 3
THEN CALL ENTER (PACK(C, S1, S2, S3), K);
/* CASE 2 -- TRIPLE CONFLICT */
DO;
TR = PACK (C, S1, S2, S3);
DO M = 0 TO CTrip;
IF SORT#(M) = TR
THEN CALL ENTER (PACK(M, P, STACK(1),
STACK(SP+1)), K);
END;
END;
END;
END;
DO CASE CX;
/* CASE 0 */

```



```

DO; DO I = 1 TO NT;
    IF IS_HEAD (GOAL_SYMBOL,I)
    THEN CALL SET (TERMINATOR,I,VALUE(TRUE));
END;
CALL SET (GOAL_SYMBOL,TERMINATOR,VALUE(FALSE));
CALL PRINT_MATRIX ('C1 MATRIX FOR STACKING DECISION',1);
CALL PRINT_TIME;
END;

/* CASE 1 */

DO; CALL PRINT_TRIPLES ('C1 TRIPLES FOR STACKING DECISION');
    IF COUNT(3) = 0 | ITERATION_COUNT > 1
    THEN RETURN;
    IF CONTROL (BYTE('I')) THEN
    IF CONTROL (BYTE('P')) | CONTROL (BYTE('O'))
    THEN RETURN;
    CALL PRINT_TIME;
    DO I = 1 TO NTRIP;
        IF TV(I) = 3
        THEN DO;
            SORT#(CTRIP) = TRIPLE(I);
            CTRIP = CTRIP + 1;
        END;
    END;
    CTRIP = CTRIP - 1;
    NETRY,NTRIP = 0;
    DOUBLE_SPACE;
    OUTPUT = 'ANALYSIS OF (2,1) CONFLICTS:';
END;

/* CASE 2 */

DO; J = 1;
    DO M = 0 TO CTRIP;
        DO K = 0 TO 1;
            I = SORT#(M);
            OUTPUT = ' ';
            THE TRIPLE : || V(SHR(I,16)) || ' ' ||
            V(SHR(I,8)&"FFF") || ' ' || V(I&"FFF") ||
            ' MUST HAVE THE VALUE : || PRINT(VALUE(K)) ||
            ' FOR ' ;
            OUTPUT = ' ';
            L = SHL(M+1,24);
        END;
    END;

```



```

TJ = HEAD(IJ);
TK = HEAD(IK);
DO WHILE (TJ & "FFF") = (TK & "FFF") & TJ ->= 0;
  TJ = SHR(TJ,8);
  TK = SHR(TK,8);
END;
IF TK = 0
  THEN DO;
    OUTPUT = '';
    OUTPUT = ' || IND1(IJ)-IND(IJ)+1 ||'
    , AND ' || IND1(IK) - IND(IK) + 1 ||'
    , VALID CONTEXTS, RESPECTIVELY, FOR';
    CALL OUTPUT_PRODUCTION(IJ);
    CALL OUTPUT_PRODUCTION(IK);
    PROPER = TJ ->= 0;
    IF PROPER
      THEN DO;
        JCL = SHL(TJ & "FFF",8);
        DO I = 1 TO NSY;
          ON_RIGHT(I) = FALSE;
        END;
      END;
    ELSE P_SAVE(IJ) = 1;
    MP = 0;
    DO PJ = IND(IJ) TO IND1(IJ);
      JCT = FILL(PJ) & "FFFF";
      JCR = JCT & "00FF";
      IF PROPER
        THEN JCT = JCL | JCR;
      ELSE JCL = JCT & "FF00";
      DO PK = IND(IK) TO IND1(IK);
        KCT = FILL(PK) & "FFFF";
        IF KCT = JCT
          THEN DO;
            IF MP < 4
              THEN CALLINGUISHED WITH (1,1) CONTEXT.';
            MP = MP | 4;
            IF PROPER
              THEN DO;
                ON_RIGHT(JCR) , || V(LEFT_PART(IK)) ||
                THEN OUTPUT = , || V(SHR(JCL,8)) ||
                , HAS , || V(JCR) || AS CONTEXT AND ,
                DOTS || V(JCR) || , IS VALID RIGHT CONTEXT FOR ,
                || V(LEFT_PART(IJ));
                ON_RIGHT(JCR) = TRUE;
              END;
            END;
          END;
        END;
      END;
    END;
  END;

```



```

ELSE OUTPUT = ' THEY HAVE EQUAL RIGHT PARTS ' ||
AND THE COMMON CONTEXT ' || V(SHR(JCL,8)) ||
DOTS || V(JCR);
CALL ADD_TROUBLE (SHR(KCT,8),LEFT_PART(IK));
END;
ELSE IF (KCT & "FF00") = JCL
THEN MP = MP | 1;
ELSE IF (KCT & "00FF") = JCR
THEN MP = MP | 2;
END;
IF MP < 4
THEN DO;
IF PROPER & (~MP) THEN OUTPUT = ' THEY CAN BE RESOLVED BY LENGTH.';
ELSE DO;
MP_SAVE(IJ) = MP_SAVE(IJ) | MP;
OUTPUT = ' THEY CAN BE RESOLVED BY ' ||
CONTEXT_CLASS(MP) || ' CONTEXT.';
END;
END;
K = K + 1;
IK = INDEX(K);
END;
EJECT PAGE;
OUTPUT = 'C2 PRODUCTION CHOICE FUNCTION:.';
TK = 0;
DO J = 1 TO NPR;
IJ = INDEX(J);
TJ = TAIL(IJ);
IF TJ = TK
THEN DO;
TK = TJ;
DOUBLE_SPACE;
OUTPUT = ' || V(TJ) || ' AS STACK TOP WILL CAUSE PRODUCTIONS ' ||
'TO BE CHECKED IN THIS ORDER:.';
END;
OUTPUT = '...';
CALL OUTPUT_PRODUCTION (IJ);
DO CASE MP_SAVE(IJ) & 3;
/* CASE 0 */
IF P_SAVE(IJ)
THEN GO TO CASE_1;
ELSE OUTPUT = '...';
THERE WILL BE NO CONTEXT CHECK.';

```



```

/* CASE 1 */
CASE_1:
DO; OUTPUT = (0,1) CONTEXT WILL BE CHECKED.  LEGAL RIGHT CONTEXT:'';
DO I = 1 TO NSY;
ON_RIGHT(I) = FALSE;
END;
DO PJ = IND(IJ) TO IND1(IJ);
JCR = F11(PJ) & "FF";
IF ON_RIGHT(JCR) THEN DO;
ON_RIGHT(JCR) = TRUE;
OUTPUT = X12 || DOTS || V(JCR);
END;
END;

/* CASE 2 */
DO; OUTPUT = (1,0) CONTEXT WILL BE CHECKED.  LEGAL LEFT CONTEXT:'';
DO I = 1 TO NSY;
ON_LEFT(I) = FALSE;
END;
DO PJ = IND(IJ) TO IND1(IJ);
JCL = SHR(F11(PJ) & "FF00",8);
IF ON_LEFT(JCL) THEN DO;
ON_LEFT(JCL) = TRUE;
OUTPUT = X12 || V(JCL) || DOTS;
END;
END;

/* CASE 3 */
DO; OUTPUT = (1,1) CONTEXT WILL BE CHECKED.  LEGAL CONTEXT:'';
DO PJ = IND(IJ) TO IND1(IJ);
OUTPUT = X12 || V(SHR(F11(PJ) & "FF00",8)) ||
DOTS || V(F11(PJ) & "FF");
END;
END;

```


PUNCH PRODUCTIONS

122


```

DO I = 1 TO L;
  CALL BUILD_CARD (J || ' ');
  DO WHILE LENGTH(V(J)) = I;
    J = J + 1;
  END;
END;
CALL PUNCH_CARD (NT+1 || ' ');
IF NT <= 15;
  THEN WIDE = 16;
  ELSE WIDE = NT;
  I = 2*WIDE + 2;
  CALL PUNCH_CARD ('DECLARE C1(NSY) BIT(' || I || ') INITIAL (');
  DO I = 0 TO NSY;
    T = '(2)';
    DO J = 0 TO WIDE;
      IF J MOD 5 = 0
      THEN DO;
        CALL BUILD_CARD (T);
        T = ',';
      END;
      T = T || GET (I,J);
    END;
    IF I < NSY
    THEN CALL PUNCH_CARD (T || ' ');
    ELSE CALL PUNCH_CARD (T || ' ');
  END;
  K = COUNT(1) - 1;
  IF K < 0
  THEN DO;
    CALL PUNCH_CARD ('DECLARE NCITRIPLES LITERALLY ''0''');
    CALL PUNCH_CARD ('DECLARE CITRIPLES(0) FIXED;');
  END;
  ELSE DO;
    CALL PUNCH_CARD ('DECLARE NCITRIPLES LITERALLY '' ' || K || ' ''');
    CALL BUILD_CARD ('DECLARE CITRIPLES(NCITRIPLES) FIXED INITIAL (');
    J = 0;
    DO I = 1 TO NTRIP;
      IF TV(I) = 1
      THEN DO;
        IF J = K
        THEN CALL PUNCH_CARD (TRIPLE(I) || ' ');
        ELSE CALL BUILD_CARD (TRIPLE(I) || ' ');
        J = J + 1;
      END;
    END;
  END;
END;
CALL BUILD_CARD ('DECLARE PRTB(' || NPR || ') FIXED INITIAL (0,');
DO I = 1 TO NPR - 1;

```



```

CALL BUILD_CARD (HEAD(INDEX(I)) || ',');
END;
PUNCH_CARD (HEAD(INDEX(NPR)) || ');');
CALL BUILD_CARD ('DECLARE PRDTB(' || NPR || ', ' || BIT(8) INITIAL (0,');
DO I = 1 TO NPR;
  L = INDEX(I);
  IF L > BASIC_NPR
    THEN L = 0;
  IF I < NPR
    THEN CALL BUILD_CARD (L || ');');
  ELSE CALL PUNCH_CARD (L || ');');
END;
BUILD_CARD ('DECLARE HDTB(' || NPR || ', ' || BIT(8) INITIAL (0,');
CALL I = 1 TO NPR - 1;
CALL BUILD_CARD (LEFT_PART(INDEX(I)) || ',');
END;
PUNCH_CARD (LEFT_PART (INDEX(NPR)) || ');');
CALL BUILD_CARD ('DECLARE PRLNGTH(' || NPR || ', ' || BIT(8) INITIAL (0,');
DO I = 1 TO NPR;
  J = I;
  K = HEAD(INDEX(I));
  DO WHILE K /= 0;
    J = J + 1;
    K = SHR(K,8);
  END;
  IF I = NPR
    THEN CALL PUNCH_CARD (J || ',');
  ELSE CALL BUILD_CARD (J || ',');
END;
BUILD_CARD ('DECLARE CONTEXT_CASE(' || NPR || ', ' || BIT(8) INITIAL (0,');
DO I = 1 TO NSY;
  TP_SAVE(I) = 0;
END;
DO I = 1 TO NPR;
  J = MP_SAVE(INDEX(I));
  K = LEFT_PART(INDEX(I));
  DO CASE J;
    /* CASE 0 */
      J = P_SAVE(INDEX(I));
    /* CASE 1 */
    ;
    /* CASE 2 */
  
```



```

TP_SAVE(K) = TP_SAVE(K) | 1;
/* CASE 3 */
TP_SAVE(K) = TP_SAVE(K) | 2;
END;
TOKEN_SAVE(K) = INDEX(I);
IF I = NPR
THEN CALL PUNCH_CARD (J || ',');
ELSE CALL BUILD_CARD (J || ',');
END;
J = 0;
DO I = NT + 1 TO NSY;
  DO SORT#(I) = J;
  IF TP_SAVE(I)
  THEN DO;
    K = TOKEN_SAVE(I);
    M = 0;
    DO L = IND(K) TO IND1(K);
      P = SHR(F11(L), 8) & "FF";
      IF P = M
      THEN DO;
        WORK(J), M = P;
        J = J + 1;
      END;
    END;
  END;
  IF J = 0
  THEN J = 1;
  CALL BUILD_CARD ('DECLARE LEFT_CONTEXT(' || J - 1 ||
  '), BIT(8) INITIAL ('');
  DO I = 0 TO J - 2;
    CALL BUILD_CARD (WORK(I) || ',');
  END;
  CALL PUNCH_CARD (WORK(J-1) || ',');
  IF J > 255
  THEN K = 16;
  ELSE K = 8;
  CALL BUILD_CARD ('DECLARE LEFT_INDEX(' || NSY-NT || ' ) BIT(' || K
  ' ) INITIAL ('');
  DO I = NT + 1 TO NSY;
    CALL BUILD_CARD (SORT#(I) || ',');
  END;
  CALL PUNCH_CARD (J || ',');
  J = 0;
  DO I = NT + 1 TO NSY;

```



```

SORT#(I) = J;
IF SHR(TP_SAVE(I),1)
THEN DO;
  K = TOKEN_SAVE(I);
  DO L = IND(K) TO IND1(K);
    TRIPLE(J) = F11(L) & "FFFF";
    J = J + 1;
  END;
END;
END;
IF J = 0
THEN J = 1;
CALL BUILD_CARD ('DECLARE CONTEXT_TRIPLE(' || J-1 || ') FIXED INITIAL (');
DO I = 0 TO J-2; (TRIPLE(I) || ',');
CALL BUILD_CARD (TRIPLE(J-1) || ',');
END;
CALL PUNCH_CARD (TRIPLE(J-1) || ',');
IF J > 255
THEN K = 16;
ELSE K = 8;
CALL BUILD_CARD ('DECLARE TRIPLE_INDEX(' || NSY-NT || ') BIT(' || K
|| ') INITIAL (');
DO I = NT + 1 TO NSY;
CALL BUILD_CARD (SORT#(I) || ',');
END;
CALL PUNCH_CARD (J || ',');
DO I = 0 TO NSY;
P_SAVE(I) = 0;
END;
DO I = 1 TO NPR;
P = 1 TO NPR;
IF P = TAIL(INDEX(I));
IF P_SAVE(P) = 0
THEN P_SAVE(P) = I;
END;
P_SAVE(NSY+1) = NPR + 1;
DO J = 0 TO NSY - 1;
I = NSY - J;
IF P_SAVE(I) = 0
THEN P_SAVE(I) = P_SAVE(I+1);
END;
CALL BUILD_CARD ('DECLARE PR_INDEX(' || NSY || ') BIT(8) INITIAL (');
DO I = 1 TO NSY;
CALL BUILD_CARD (P_SAVE(I) || ',');
END;
CALL PUNCH_CARD(NPR+1 || ',');
CALL PRINT_TIME;
END PUNCH_PRODUCTIONS;

```


THE SEMANTIC PREPROCESSOR

THE FOLLOWING CARDS ARE PUNCHED BY THE SYNTAX PRE-PROCESSOR

129

131

132

134

135

136

[illegible]

DECLARATIONS FOR THE SCANNER

[illegible]

```

DECLARE
DOUBLE_SPACE LITERALLY 'OUTPUT(1) = DOUBLE', /* PRINTER CONTROL
EJECT_PAGE LITERALLY 'OUTPUT(1) = PAGE', /* PRINTER CONTROL
FALSE LITERALLY '0', /* GLOBAL LITERAL

```


[illegible]


```

      "FFFFF", "FFFFFFF"),
RESERVED LIMIT FIXED,
S CHARACTER,
SEVERE ERRORS FIXED,
SSP FIXED,
STOPT(100) BIT(1),
TEXT CHARACTER,
TEXT LIMIT FIXED,
TOKEN FIXED,
TX(255) BIT(8),
XVAR(STACKSIZE) CHARACTER;',
XX1 CHARACTER INITIAL (''),
XX4 CHARACTER INITIAL (''),
XX70 CHARACTER INITIAL ('');

```

DECLARATIONS FOR THE SEMANTIC META-LANGUAGE


```

ELSE RETURN STRING || SUBSTR(X70,0,WIDTH-L);
END PAD;

I_FORMAT:
PROCEDURE (NUMBER,WIDTH) CHARACTER;
DECLARE (NUMBER,WIDTH,L) FIXED, STRING CHARACTER;
STRING = NUMBER;
L = LENGTH(STRING);
IF L >= WIDTH
THEN RETURN STRING;
ELSE RETURN SUBSTR(X70,0,WIDTH-L) || STRING;
END I_FORMAT;

ERROR:
PROCEDURE (MSG,SEVERITY);
DECLARE MSG CHARACTER, SEVERITY FIXED;
ERROR_COUNT = ERROR_COUNT + 1;
IF CONTROL(BYTE('L')) & SAVED ^= NUL
THEN DO;
    OUTPUT = I_FORMAT(CARD_COUNT,4) || ' ' || ' ' || SAVED || ' ' || ' ' || SAVED;
    SAVED = NUL;
END; ELSE
IF CONTROL(BYTE('L'))
THEN OUTPUT = I_FORMAT(CARD_COUNT,4) || ' ' || ' ' || BUFFER || ' ' || ' ';
OUTPUT = SUBTPointer, TEXT LIMIT-CP+MARGIN-CHOP);
OUTPUT = '*** ERROR, ' || MSG || ' ' || LAST PREVIOUS ERROR WAS ' ||
        'DETECTED ON LINE ' || PREVIOUS_ERROR || ' ' || '***';
PREVIOUS_ERROR = CARD_COUNT;
IF SEVERITY > 0
THEN IF SEVERE_ERRORS = 0
THEN DO;
    OUTPUT = '*** TOO MANY SEVERE ERRORS, CHECKING ABORTED ***';
    COMPILING = FALSE;
END; ELSE SEVERE_ERRORS = SEVERE_ERRORS + 1;
END ERROR;

ERRORUP:
PROCEDURE;
CALL ERROR('DUPLICATE IDENTIFIER',1);
END ERRORUP;

ERRORUN:
PROCEDURE;
CALL ERROR('UNDECLARED IDENTIFIER',1);
END ERRORUN;

ENTER:

```



```

PROCEDURE(ENTRY) FIXED;
DECLARE ENTRY CHARACTER;
SYM_PTR = SYM_PTR + 1;
IF SYM_PTR > MAXSYM
THEN DO;
CALL ERROR('SYMBOL TABLE OVERFLOW',1);
CALL EXIT;
END;
SYMBOL(SYM_PTR) = ENTRY;
TYPE(SYM_PTR) = 0;
ATT1(SYM_PTR) = 0;
RETURN SYM_PTR;
END ENTER;

LOOKUP:
PROCEDURE(ENTRY) FIXED;
DECLARE ENTRY CHARACTER, I FIXED;
DO I = 1 TO SYM_PTR;
IF SYMBOL(I) = ENTRY
THEN RETURN I;
END;
RETURN 0;
END LOOKUP;

CHECKCHAR:
PROCEDURE(S);
DECLARE S CHARACTER, I FIXED;
IF S = SUBSTR(ST5,EIGHTY,1)
THEN RETURN 0;
IF S = SUBSTR(ST2,QUOT,1)
THEN RETURN 1;
DO I = 0 TO 9;
IF S = SUBSTR('0123456789',I,1)
THEN RETURN 2;
END;
RETURN 3;
END CHECKCHAR;

CHECKCODE:
PROCEDURE(S) BIT(1);
DECLARE S CHARACTER, (I,J,K,TEST,VAR) FIXED;
DO WHILE SUBSTR(S,0,1) = X1;
S = SUBSTR(S,1);
END;
I = LENGTH(S);
J = I - 1;
DO WHILE SUBSTR(S,J,1) = X1;
S = SUBSTR(S,0,J);

```



```

J = LENGTH(S) - 1;
END;
I = J + 1;
VAR = BYTE(S);
DO CASE CHECKCHAR(SUBSTR(S,0,1));
/* CASE 0 - BIT STRING */
DO J = 1 TO I - 1;
IF BYTE(S,J) = VAR THEN
IF J = I - 1
THEN RETURN TRUE;
ELSE RETURN FALSE;
END;
/* CASE 1 - CHARACTER STRING */
DO J = 1 TO I - 1;
IF BYTE(S,J) = BYTE('') THEN
IF J = I - 1
THEN RETURN TRUE;
ELSE RETURN FALSE;
END;
/* CASE 2 - NUMBER */
DO; DO J = 1 TO I - 1;
TEST = 0;
DO K = 0 TO 9;
IF BYTE(S,J) = BYTE('0123456789',K)
THEN TEST = 1;
END;
IF TEST = 0
THEN RETURN FALSE;
END;
RETURN TRUE;
END;
/* CASE 3 - VARIABLE */
DO; TEST = 0;
DO J = 1 TO I - 1;
IF BYTE(S,J) = BYTE('') THEN
IF TEST = 0
THEN TEST = J;
END;

```



```

IF TEST > 0
  THEN S = SUBSTR(S,0,TEST);
  IF LOOKUP(S) = 0 THEN
    IF S = 'VAR' & S = 'FIXV'
      THEN RETURN FALSE;
    RETURN TRUE;
END;

```

```
END; CHECKCODE;
```



```

T = SUBSTR(ST4,AI2,2) || SUBSTR(ST3,EQ+1,1);
T = S1 || SUBSTR(ST2,PTR,4);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST4,POINT,7) || S2;
T = SUBSTR(ST2,RP,1);
T = SUBSTR(ST3,EQ+1,1);
T = SUBSTR(ST4,AI2,2);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = S1 || SUBSTR(ST2,PTR,4);
T = SUBSTR(ST3,EQ+1,1);
T = S1 || SUBSTR(ST2,PTR,4);
T = SUBSTR(ST4,P1,2);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
END TABLESET;

STACKSET:
PROCEDURE (S1,S2,S3); CHARACTER;
DECLARE (S1,S2,S3,T);
T = SUBSTR(ST2,FI,3) || S1;
T = SUBSTR(ST2,PTR,4);
T = SUBSTR(ST3,GT,3) || S1;
T = SUBSTR(ST2,EX,3);
CALL EMIT(T);
T = SUBSTR(ST2,TD,9) || 'CALL ERROR(''STACK OVERFLOW'');';
CALL SUBSTR(ST2,XIT,10) || SUBSTR(ST2,FIN,4);
CALL EMIT(T);
T = S1 || SUBSTR(ST3,EQ+1,1);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = S1 || SUBSTR(ST4,SPTR,7) || S3;
T = SUBSTR(ST2,RP,1);
T = SUBSTR(ST3,EQ+1,1);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = S1 || SUBSTR(ST2,PTR,4);
T = SUBSTR(ST3,EQ+1,1);
T = S1 || SUBSTR(ST2,PTR,4);
T = SUBSTR(ST4,P1,2);
T = SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
END STACKSET;

```



```

/* CASE 48 */
<TAG DECLARATION> ::= <TAG DECLARATION> , <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = TAG_ID;
    T = S | SUBSTR(ST2,LIT,12) | TAG_COUNT;
    T = T | SUBSTR(ST2,QUOT,1);
    CALL FMIT(T | X1);
    TAG_COUNT = TAG_COUNT + 1;
  END;
  ELSE CALL ERRORDUP;
END;

/* CASE 49 */
<STRING DECLARATION> ::= STRING <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = STRING_ID;
    CALL FMIT(SUBSTR(ST2,LP,1) | S);
  END;
  ELSE CALL ERRORDUP;
END;

/* CASE 50 */
<STRING DECLARATION> ::= <STRING DECLARATION> , <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = STRING_ID;
    CALL FMIT(SUBSTR(ST2,COMMA,1) | S);
  END;
  ELSE CALL ERRORDUP;
END;

/* CASE 51 */
<FLAG DECLARATION> ::= FLAG <IDENTIFIER> */

```



```

END;

/* CASE 45 */
<CELL DECLARATION> ::= CELL <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = CELL_ID;
    CALL EMIT(SUBSTR(ST2,LP,1) || S);
  END;
  ELSE CALL ERRORRUP;
END;

/* CASE 46 */
<CELL DECLARATION> ::= <CELL DECLARATION> , <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = CELL_ID;
    CALL EMIT(SUBSTR(ST2,COMMA,1) || S);
  END;
  ELSE CALL ERRORRUP;
END;

/* CASE 47 */
<TAG DECLARATION> ::= TAG <IDENTIFIER> */

DO;
  S = VAR(SP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = TAG_ID;
    T = S || SUBSTR(ST2,LP,12) || TAG_COUNT;
    T = T || SUBSTR(ST2,QUOT,1);
    CALL EMIT(T || X1);
    TAG_COUNT = TAG_COUNT + 1;
  END;
  ELSE CALL ERRORRUP;
END;

```



```

DO;
END;

/* CASE 38 */
<TYPE SPECIFIER> ::= CHAR    */

DO;
    FIXV(MP) = TYPE_CHAR;
END;

/* CASE 39 */
<TYPE SPECIFIER> ::= INTEGER    */

DO;
    FIXV(MP) = TYPE_INTEGER;
END;

/* CASE 40 */
<BIT> ::= BIT    */

DO;
    FIXV(MP) = TYPE_BIT | SHL(32,8);
END;

/* CASE 41 */
<BIT> ::= BIT <NUMBER>    */

DO;
    FIXV(MP) = TYPE_BIT | SHL(FIXV(SP),8);
END;

/* CASE 42 */
<STACK DECLARATION> ::= <STACK> <STABLE SPECIFIER>    */

DO;
    END;

/* CASE 43 */
<STACK DECLARATION> ::= <STACK DECLARATION> , <STABLE SPECIFIER>    */

DO;
    END;

/* CASE 44 */
<STACK> ::= STACK    */

DO;
    STACK_FLAG = TRUE;

```



```

/* CASE 35 */
<FIELD> ::= <FIELD> , <FIELD> */
DO;
END;

/* CASE 36 */
<FIELD> ::= <IDENTIFIER> <TYPE SPECIFIER> */
DO;
  S = VAR(MP);
  I1 = FIXV(SP);
  I2 = I1 & "FF";
  I3 = SHR(I1,8);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    IF TABLE_FLAG PTR) = TABLE_ENTRY;
    ELSE TYPE(SYM_PTR) = STACK_ENTRY;
    AT1(SYM_PTR) = I2;
    IF EXTENT > 0
    THEN DO;
      T = S || SUBSTR(ST2,LP,1) || EXTENT;
      T = T || SUBSTR(ST2,RP,1) || X1;
    END; ELSE
    IF TABLE_FLAG
    THEN T = S || SUBSTR(ST2,MAXKEY,9);
    ELSE T = S || SUBSTR(ST2,MAXSTK,9);
    IF I3 > 0
    THEN DO;
      T = T || TYPE(I2) || SUBSTR(ST2,LP,1);
      T = T || I3 || SUBSTR(ST2,RP,1);
      T = T || SUBSTR(ST2,COMMA,1) || X1;
    END;
  ELSE DO;
    T = T || TYPE(I2) || SUBSTR(ST2,COMMA,1);
    T = T || X1;
  END;
  CALL EMIT(T);
END;
ELSE CALL ERRORDUP;
END;

/* CASE 37 */
<TYPE SPECIFIER> ::= <BIT> */

```



```

T = S || SUBSTR(ST2, PTR, 4) ;
T = T || SUBSTR(ST2, FIX+1, 6) ;
T = T || SUBSTR(ST2, INIT, 10) ;
T = T || SUBSTR(ST2, RP, 1) ;
CALL EMIT(T) ; SUBSTR(ST2, COMMA, 1) || X1 ;
T = S || SUBSTR(ST2, EX, 3) ;
T = T || SUBSTR(ST2, FIX+1, 6) ; EXTENT ;
T = T || SUBSTR(ST2, INIT, 9) ;
T = T || SUBSTR(ST2, RP, 1) ;
T = T || SUBSTR(ST2, COMMA, 1) || X1 ;
CALL EMIT(T) ;
END ;
T = S || SUBSTR(ST2, LIT, 12) ;
T = T || TAG_COUNT || SUBSTR(ST2, QUOT, 1) ;
CALL EMIT(T) ; SUBSTR(ST2, COMMA, 1) || X1 ;
TAG_COUNT = TAG_COUNT + 1 ;
IF STACK_FLAG
THEN DO :
T = S || SUBSTR(ST2, TOP, 4) ;
T = T || SUBSTR(ST2, FIX+1, 6) ;
T = T || SUBSTR(ST2, INIT, 10) ;
T = T || SUBSTR(ST2, RP, 1) ;
CALL EMIT(T) || SUBSTR(ST2, COMMA, 1) || X1 ;
END ;
END ;
ELSE CALL ERRORUP ;
END ;
/* CASE 32 */
<NUMBER PART> ::= ( <NUMBER> : */
DO ;
FIXV(MP) = FIXV(MPP1) ;
END ;
/* CASE 33 */
<NUMBER PART> ::= ( $ : */
DO ;
FIXV(MP) = 0 ;
END ;
/* CASE 34 */
<FIELDS> ::= <FIELD> */
DO ;
END ;

```



```

END;

/* CASE 26 */
<DECLARATION> ::= <FLAG DECLARATION> */

DO;
END;

/* CASE 27 */
<TABLE DECLARATION> ::= <TABLE> <STABLE SPECIFIER> */

DO;
END;

/* CASE 28 */
<TABLE DECLARATION> ::= <TABLE DECLARATION> , <STABLE SPECIFIER> */

DO;
END;

/* CASE 29 */
<TABLE> ::= TABLE */

DO;
TABLE_FLAG = TRUE;
END;

/* CASE 30 */
<STABLE SPECIFIER> ::= <STABLE HEADER> <FIELDS> ) */

DO;
END;

/* CASE 31 */
<STABLE HEADER> ::= <IDENTIFIER> <NUMBER PART> */

DO;
  S = VAR(MP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    IF TABLE_FLAG
    THEN TYPE(SYM_PTR) = TABLE_ID;
    ELSE TYPE(SYM_PTR) = STACK_ID;
    EXTENT = FIXV(SP);
    ATT1(SYM_PTR) = EXTENT;
    IF EXTENT >
    THEN DO;

```



```

/* <LABEL> ::= <IDENTIFIER> : */
DO;
  S = VAR(MP);
  IF LOOKUP(S) = 0
  THEN DO;
    CALL ENTER(S);
    TYPE(SYM_PTR) = TYPE LABEL;
    CALL EMIT(S || SUBSTR(ST2,COL,1));
  END;
  ELSE CALL ERRORUP;
END;

/* CASE 21 */
/* <DECLARATION> ::= <TABLE DECLARATION> */
DO;
  TABLE_FLAG = FALSE;
END;

/* CASE 22 */
/* <DECLARATION> ::= <STACK DECLARATION> */
DO;
  STACK_FLAG = FALSE;
END;

/* CASE 23 */
/* <DECLARATION> ::= <CELL DECLARATION> */
DO;
  CALL EMIT(SUBSTR(ST2,FIX,9));
END;

/* CASE 24 */
/* <DECLARATION> ::= <TAG DECLARATION> */
DO;
  END;
END;

/* CASE 25 */
/* <DECLARATION> ::= <STRING DECLARATION> */
DO;
  T = SUBSTR(ST2,RP,1) || X1;
  T = T || TYPE(1);
  T = T || SUBSTR(ST2,COMMA,1) || X1;
  CALL EMIT(T);

```



```

/* <SENTENCE> ::= <SENTENCE> <STATEMENT>    */
DO;
END;

/* <STATEMENT> ::= <UNCONDITIONAL> ;    */
DO;
END;

/* <STATEMENT> ::= <CONDITIONAL>    */
DO;
END;

/* <STATEMENT> ::= <LABEL> <STATEMENT>    */
DO;
END;

/* <STATEMENT> ::= <COMMENT>    */
DO;
END;

/* <COMMENT> ::= <LDEL> <REMARKS> <RDEL>    */
DO;
END;

/* <LDEL> ::= /* */
DO;
END;

/* <RDEL> ::= !STAR! */
DO;
END;

/* CASE 20 */

```



```

S = S || SUBSTR(ST2,KASE2,7);
OUTBUF(OUTCOUNT) = S;
OUTCOUNT = LENGTH(S);
END;

```

```

/* CASE 10 */
<DELIMITER 2> ::= a */

```

```

DO; T = SUBSTR(ST2,FIN,4);
IF I1 = 80 - OUTCOUNT;
IF I1 < 4
THEN DO;
S = OUTBUF(OUTCOUNT);
S = S || SUBSTR(X80,0,I1);
OUTBUF(OUTCOUNT) = S;
OUTCOUNT = OUTCOUNT + 1;
S = X4 || T;
S = S || SUBSTR(X80,0,72);
OUTBUF(OUTCOUNT) = S;
END; ELSE
IF I1 = 4
THEN CALL EMIT(T);
ELSE CALL EMIT(T || SUBSTR(X80,0,I1-4));
DOUBLE SPACE;
IF CONTROL(BYTE('C'))
THEN DO I1 = 0 TO OUTCOUNT;
OUTPUT = SUBSTR(ST2,CARDOUT,35) || OUTBUF(I1);
END;
IF CONTROL(BYTE('P'))
THEN DO I1 = 0 TO OUTCOUNT;
OUTPUT(PUNCH) = OUTBUF(I1);
END;
DOUBLE SPACE;
IF CONTROL(BYTE('L')) & CONTROL(BYTE('C'))
THEN OUTPUT = I_FORMAT(CARD_COUNT,4) || ' ' || SAVED || ' ' || SAVED;
SAVED = ' ';
OUTCOUNT = 0;
END;

```

```

/* CASE 11 */
<SENTENCE> ::= <STATEMENT> */

```

```

DO;
END;

```

```

/* CASE 12 */

```



```

/* CASE 3 */
<BEGIN> ::= BEGIN */

DO;
  OUTBUF(BUFCOUNT) = SUBSTR(ST1,DECL,80);
  OUTCOUNT=80;
  CALL EMIT(SUBSTR(ST2,DECL,8));
END;

/* CASE 4 */
<DECLARATION SET> ::= <DECLARATION> ; */

DO;
  END;

/* CASE 5 */
<DECLARATION SET> ::= <DECLARATION SET> <DECLARATION> ; */

DO;
  END;

/* CASE 6 */
<SEQUENCE> ::= <BRACKETED SENTENCE> */

DO;
  END;

/* CASE 7 */
<SEQUENCE> ::= <SEQUENCE> <BRACKETED SENTENCE> */

DO;
  END;

/* CASE 8 */
<BRACKETED SENTENCE> ::= <DELIMITER 1> <SENTENCE> <DELIMITER 2> */

DO;
  END;

/* CASE 9 */
<DELIMITER 1> ::= # */

DO;
  CASE COUNT = CASE_COUNT + 1;
  IF CASE_COUNT = 1
  THEN DO;
    S = SUBSTR(ST2,KASE1,9) || CASE_COUNT;

```



```

PROCEDURE( PRODUCTION_NUMBER );
DECLARE PRODUCTION_NUMBER FIXED;
DO CASE PRODUCTION_NUMBER;

/* CASE 0 - NULL CASE */

DO;
END;

/* CASE 1 */
/* <SEMANTIC PROGRAM> ::= <HEADER> <SEQUENCE> END */
DO;
IF MP = 2
THEN DO;
CALL ERROR('EOF AT INVALID POINT',1);
CALL STACK_DUMP;
END;
COMPILING = FALSE;
END;

/* CASE 2 */
/* <HEADER> ::= <BEGIN> <DECLARATION SET> */
DO;
I1 = CUTCOUNT - 2;
I2 = 80 - I1;
S = SUBSTR(OUTBUF(BUFCOUNT),C,I1);
S = S || SUBSTR(ST2,SEMI,1);
S = S || SUBSTR(X80,0,I2);
OUTBUF(BUFCOUNT) = S;
DOUBLE_SPACE;
IF CONTROL(BYTE('C'))
THEN DO I1 = 0 TO BUFCOUNT;
OUTPUT = SUBSTR(ST2,CARDOUT,35) || OUTBUF(I1);
END;
IF CONTROL(BYTE('P'))
THEN DO I1 = 0 TO BUFCOUNT;
OUTPUT(PUNCH) = OUTBUF(I1);
END;
DOUBLE_SPACE;
IF CONTROL(BYTE('L')) & CONTROL(BYTE('C'))
THEN OUTPUT = I_FORMAT(CARD_COUNT,4) || ' ' || SAVEB || ' ' || SAVER;
SAVEB = ' ';
BUFCOUNT = 0;
OUTBUF(BUFCOUNT) = SUBSTR(ST1,SYNT,80);
OUTCOUNT = 80;
END;

```



```

IF S = '/' ;
THEN DIVIDE = 1 ; ELSE
IF S = '|' ;
THEN EOFFILE = 1 ; ELSE
IF S = ':' ;
THEN STOPIT(1) = TRUE ; ELSE
IF S = '*' ;
THEN MULT = 1 ; ELSE
IF S = '**' ;
THEN EXPON = 1 ; ELSE
IF S = '<BIT STRING>' ;
THEN BITSTRING = 1 ; ELSE
IF S = '<CHARACTER STRING>' ;
THEN CHARSTRING = 1 ;
END ;
IF IDENT = NT
THEN RESERVED_LIMIT = LENGTH(V(NT-1)) ;
ELSE RESERVED_LIMIT = LENGTH(V(NT)) ;
STOPIT(EOFFILE) = EOF ;
DO I = 0 TO 255 ;
NOT_LETTER_OR_DIGIT(I) = TRUE ;
END ;
DO I = 0 TO LENGTH(ALPHABET) - 1 ;
J = BYTE(ALPHABET,I) ;
TX(J) = 1 ;
NOT_LETTER_OR_DIGIT(J) = FALSE ;
CHARTYPE(J) = 1 ;
END ;
DO I = 0 TO 9 ;
J = BYTE('0123456789',I) ;
NOT_LETTER_OR_DIGIT(J) = FALSE ;
CHARTYPE(J) = 2 ;
END ;
DO I = V_INDEX(0) TO V_INDEX(1) - 1 ;
J = BYTE(V(I)) ;
TX(J) = 1 ;
CHARTYPE(J) = 3 ;
END ;
CHARTYPE(BYTE(' ')) = 4 ;
CHARTYPE(BYTE('*')) = 5 ;
CHARTYPE(BYTE('/')) = 6 ;
CHARTYPE(BYTE('"')) = 7 ;
CHARTYPE(BYTE('\'')) = 8 ;
CHARTYPE(BYTE(':')) = 9 ;
CP = 0 ;
TEXT_LIMIT = -1 ;
TEXT = '' ;

```



```
THEN MESSAGE = MESSAGE || 'C';
OUTPUT = MESSAGE || ' ';
END PRINT_TIME;
```

```

PRINT DATE AND TIME:
PROCEDURE(MESSAGE,D,T):
DECLARE MESSAGE CHARACTER (D,T,YEAR,DAY,M) FIXED:
DECLARE MONTH(11) CHARACTER INITIAL ('JANUARY','FEBRUARY','MARCH',
'APRIL','MAY','JUNE','JULY','AUGUST','SEPTEMBER','OCTOBER',
'NOVEMBER','DECEMBER'), DAYS(12) FIXED INITIAL(0,31,60,91,121,
152,182,213,244,274,305,335,366);
YEAR = D/1000 + 1900;
DAY = D MOD 1000;
IF (YEAR < "3") THEN = 0
THEN IF DAY > 59
THEN DAY = DAY + 1;
M = 1;
DO WHILE DAY > DAYS(M);
M = M + 1;
END;
CALL PRINT_TIME(MESSAGE || MONTH(M-1) || X1 || DAY - DAYS(M-1) ||
|| YEAR || :CLOCK TIME = ,T);
END PRINT_DATE_AND_TIME;

```

[illegible]

```

INITIALIZATION:
PROCEDURE:
EJECT PAGE;
CALL PRINT DATE AND TIME('SYNTAX CHECK AND CODE PRODUCTION -- SEMANTIC ');
CALL PROCESSOR VERSION OF('DATE_OF_GENERATION,TIME_OF_GENERATION');
DOUBLE SPACE;
DO I = 1 TO NT;
  S = V(I);
  IF S = ':' THEN COLON = I; ELSE
  IF S = '=' THEN EQUALS = I; ELSE
  THEN S = '<NUMBER>'; ELSE
  IF S = 'NUMBER' THEN IDENTIFIER = I; ELSE
  IF S = '<IDENTIFIER>' THEN IDENT = I; ELSE

```



```

      THEN BCD = BCD || SUBSTR(TEXT,S1,CP-S1);
      CALL CHAR;
      IF BYTE(TEXT,CP) ^= BYTE('')
      THEN RETURN;
      IF LENGTH(BCD) > 255
      THEN DO;
        CALL ERROR('CHARACTER STRING TOO LONG',0);
        RETURN;
      END;
      BCD = BCD || '';
      TEXT_LIMIT = TEXT_LIMIT - CP;
      TEXT = SUBSTR(TEXT,CP);
      CP = 0;
    END;

/* CASE 9 -- COLON OR EQUALS */

DO;
  CP = 1;
  IF BYTE(TEXT,CP) ^= BYTE('=')
  THEN TOKEN = COLON;
  ELSE DO;
    TOKEN = EQUALS;
    CP = CP + 1;
  END;
  RETURN;
END;

END;
CP = CP + 1;
END;
SCAN;
END;

```

[illegible]


```

CP = 1;
S1 = BYTE(TEXT,CP);
DO WHILE S1 ^= BYTE('');
  IF CP > 32
    THEN DO;
      CALL ERROR('BIT STRING TOO LONG',0);
    RETURN;
  END;
  CP = CP + 1;
  IF CP > TEXT_LIMIT
    THEN CALL GET_CARD;
  S1 = BYTE(TEXT,CP);
END;
CP = CP + 1;
IF CP > 1
  THEN BCD = BCD || SUBSTR(TEXT,0,CP);
RETURN;
END;

/* CASE 8 -- CHARACTER STRING */
DO FOREVER;
  TOKEN = CHARSTRING;
  S1 = 1;
  DO WHILE CP + 1;
    IF CP <= TEXT_LIMIT
      THEN CP = CP + 1;
    ELSE DO;
      IF LENGTH(BCD) + CP > 257
        THEN DO;
          CALL ERROR('CHARACTER STRING TOO LONG',0);
        RETURN;
      END;
      IF CP > S1
        THEN BCD = BCD || SUBSTR(TEXT,S1,CP-S1);
      TEXT = X1;
      CP = 0;
      CALL GET_CARD;
      S1 = C;
    END;
  END;
  IF LENGTH(BCD) + CP > 257
    THEN DO;
      CALL ERROR('CHARACTER STRING TOO LONG',0);
    RETURN;
  END;
  CALL ERROR('CHARACTER STRING TOO LONG',0);
RETURN;
END;
IF CP > S1

```



```

END;

/* CASE 5 -- MULTIPLICATION OR EXPONENTIATION */
DO;
  CP = 1;
  IF BYTE(TEXT,CP) ^= BYTE('*')
  THEN TOKEN = MULT;
  ELSE DO;
    TOKEN = EXPON;
    CP = CP + 1;
  END;
  RETURN;
END;

/* CASE 6 -- COMMENT OR DIVIDE */
DO;
  CALL CHAR;
  IF BYTE(TEXT,CP) ^= BYTE('*')
  THEN DO;
    TOKEN = DIVIDE;
    RETURN;
  END;
  S1, S2 = BYTE(' ');
  DO WHILE S1 ^= BYTE('*') | S2 ^= BYTE('/');
    IF S1 = BYTE('$')
    THEN DO;
      CONTROL(S2) = CONTROL(S2);
      IF S2 = BYTE('T')
      THEN CALL TPAGE;
      ELSE
        IF S2 = BYTE('U')
        THEN CALL UNTRACE;
      ELSE
        IF S2 = BYTE('I')
        THEN IF CONTROL(S2)
        THEN MARGIN_CHOP = TEXT_LIMIT - CP + 1;
      ELSE MARGIN_CHOP = 0;
    END;
    S1 = S2;
    CALL CHAR;
    S2 = BYTE(TEXT,CP);
  END;
END;

/* CASE 7 -- BIT STRING */
DO;
  TOKEN = BITSTRING;

```



```

THEN IF S1 <= RESERVED_LIMIT
THEN DO I = V_INDEX(S1-1) TO V_INDEX(S1) - 1;
  IF BCD = V(I)
  THEN DO;
    TOKEN = I;
    RETURN;
  END;
END;
TOKEN = IDENT;
RETURN;
END;
END;
BCD = PCO || TEXT;
CALL GET_CARD;
CP = -1;
END;

/* CASE 2 -- NUMBER */
DO;
  TOKEN = NUMBER;
  DO FOREVER;
    DO CP = CP TO TEXT_LIMIT;
      S1 = BYTE(TEXT,CP);
      IF S1 < "FO"
      THEN RETURN;
      NUMBER_VALUE = 10 * NUMBER_VALUE + S1 - "FO";
    END;
    CALL GET_CARD;
  END;
END;

/* CASE 3 -- SPECIAL CHARACTER */
DO;
  TOKEN = TX(BYTE(TEXT));
  CP = 1;
  RETURN;
END;

/* CASE 4 -- BLANK */
DO;
  CP = 1;
  DO WHILE BYTE(TEXT,CP) = BYTE(' ') & CP <= TEXT_LIMIT;
    END;
    CP = CP + 1;
  END;
  CP = CP - 1;

```



```

DO; S = VAR(SP);
IF LOOKUP(S) = 0
THEN DO;
CALL ENTER(S);
TYPE(SYM_PTR) = FLAG_ID;
T = S || SUBSTR(ST5,BIT1,18);
T = T || SUBSTR(ST2,COMMA,1) || X1;
CALL EMIT(T);
END;
ELSE CALL ERRORUP;
END;

/* CASE 52 */
/* <FLAG DECLARATION> ::= <FLAG DECLARATION> , <IDENTIFIER> */

DO; S = VAR(SP);
IF LOOKUP(S) = 0
THEN DO;
CALL ENTER(S);
TYPE(SYM_PTR) = FLAG_ID;
T = S || SUBSTR(ST5,BIT1,18);
T = T || SUBSTR(ST2,COMMA,1) || X1;
CALL EMIT(T);
END;
ELSE CALL ERRORUP;
END;

/* CASE 53 */
/* <UNCONDITIONAL> ::= <ASSIGNMENT> */

DO;
END;

/* CASE 54 */
/* <UNCONDITIONAL> ::= <OPERATION> */

DO;
END;

/* CASE 55 */
/* <UNCONDITIONAL> ::= <TRANSFER> */

DO;
END;

```



```

/* CASE 56 */
<ASSIGNMENT> ::= <LEFT PART> <ARITHMETIC> */

DO;
S = VAR(MP);
T = VAR(SP);
I1 = LOOKUP(S);
I2 = LOOKUP(T);
IF PUSH
THEN DO;
IF LIST_PTR > C
THEN DO;
CALL EMIT('I1=STACK(' || LIST_OP(C) || ',' || S || ' ');
CALL EMIT('STKPTR(I1)= ' || T || 'TOP;');
CALL EMIT('S ' || 'TOP=I1;');
IF AT(I1) = 0
THEN CALL EMIT('STKPTR(I1)=0;I2=I1;');
ELSE CALL STACKSET(S, I2, 'I1');
DO I3 = 1 TO LIST_PTR;
I4 = I1 + I3;
IF TYPE(I4) = STACK_ENTRY
THEN CALL ERROR('TOO MANY STACK PARAMETERS',1);
ELSE CALL EMIT(SYMBOL(I4) || '(I2)= ' || LIST_OP(I3) || ' ');
END;
LIST_PTR = 0;
END; ELSE
IF TYPE(I2) = STACK_ID
THEN DO;
CALL EMIT('IF ' || T || 'TOP = 0 THEN ');
CALL EMIT('CALL ERROR(' || 'STACK EMPTY;'); ELSE DO;');
CALL EMIT('I1=STKPTR(' || T || 'TOP;');
CALL EMIT('IF I1 = C THEN I1= ' || T || 'TOP;');
CALL EMIT('S=STK(' || T || 'TOP;');
CALL EMIT('I2=STACK(S, ' || S || 'TOP;');
CALL EMIT('STKPTR(I2)= ' || S || 'TOP;');
CALL EMIT('S ' || 'TOP=I2;');
IF AT(I1) = C
THEN CALL EMIT('STKPTR(I2)=0;I3=I2;');
ELSE CALL STACKSET(S, I3, 'I2');
I3 = I1 + 1;
I4 = I2 + TYPE(I3) = STACK_ENTRY;
DO WHILE TYPE(I4) = STACK_ENTRY;
IF TYPE(I4) = STACK_ENTRY
THEN CALL ERROR('UNPAIRED STACK PARAMETERS',1);
ELSE CALL EMIT(SYMBOL(I3) || '(I3)= ' || SYMBOL(I4) || '(I1);');
I3 = I3 + 1;
I4 = I4 + 1;
END;

```



```

CALL EMIT(T || ' _TOP=STKOV(R(I1));');
IF ATT1(I1) ^= 0
THEN CALL EMIT(T || ' _PTR=' || T || ' _PTR-1;');
IF TOP_FLAG
THEN TOP_FLAG = FALSE;
ELSE CALL EMIT('CALL POP(' || T || ' _TOP);');
CALL FMIT(SUBSTR(ST2,FIN,4));
END;
ELSE CALL ERROR('INVALID STACK OPERATION',1);
PUSH = FALSE;
END; ELSE
IF LIST_FLAG
THEN DO;
IF LIST_LEVEL > 0
THEN DO;
IF LIST_PTR - LIST_LEVEL ^= LIST_LEVEL + 1
THEN CALL ERROR('UNPAIRED LIST OPERANDS',1);
ELSE DO I3 = 0 TO LIST_LEVEL;
I4 = LIST_LEVEL + I3 + 1;
CALL EMIT(LIST_OP(I3) || '=' || LIST_OP(I4) || ';' );
END;
LIST_LEVEL = 0;
END; ELSE
IF TYPE(I2) = STACK_ID | TOP_FLAG
THEN DO;
CALL EMIT('IF ' || T || ' TOP = 0 THEN ');
CALL EMIT('CALL ERROR(' || STACK_EMPTY || '); ELSE DO;');
CALL EMIT('IF I1 = U THEN I1= ' || T || ' _TOP;');
U = LIST_OP(Q) || SUBSTR(ST4,STAK-1,5);
U = U || SUBSTR(ST2,1);
U = U || SUBSTR(ST2,RP,1);
CALL EMIT(U);
DO I3 = 1 TO LIST_PTR;
I4 = I2 + I3;
IF TYPE(I4) ^= STACK_ENTRY
THEN CALL ERROR('TOO MANY LIST PARAMETERS',1);
ELSE DO;
U = LIST_OP(I3) || SUBSTR(ST3,EQ+1,1);
U = U || SYMBOL(I4) || SUBSTR(ST4,P1P,5);
CALL FMIT(U);
END;
END;
IF TOP_FLAG
THEN DO;
CALL EMIT(T || ' _TOP=STKOV(R(I1));');
IF ATT1(I1) ^= 0
THEN CALL EMIT(T || ' _PTR=' || T || ' _PTR-1;');

```



```

CALL EMIT('CALL POP(' || T || '_TOP);');
END;
ELSE TOP_FLAG = FALSE;
CALL EMIT(SUBSTR(ST2,FIN,4));
END; ELSE
IF SET_FLAG
THEN DO;
IF TABLE_PTR <= LIST_PTR + 1
THEN CALL_ERROR('UNPAIRED OPERANDS',1);
ELSE DO I3 = 0 TO LIST_PTR;
CALL EMIT(LIST_OP(I3) || '=' || TABLE_OP(I3+1) || '(' ||
TABLE_OP(0) || ');');
END;
SET_FLAG = FALSE;
TABLE_PTR = 0;
END;
LIST_PTR = 0;
LIST_FLAG = FALSE;
END; ELSE
IF LEFT SET
THEN DO;
IF LIST_PTR < 1
THEN CALL_ERROR('INVALID OPERATION',1); ELSE
IF TABLE_PTR <= LIST_PTR + 1
THEN CALL_ERROR('UNPAIRED OPERANDS',1);
ELSE DO I3 = 0 TO LIST_PTR;
CALL EMIT(TABLE_OP(I3+1) || '(' || TABLE_OP(0) || ')=' ||
LIST_OP(I3) || ');');
END;
SET = FALSE;
LIST_PTR = 0;
TABLE_PTR = 0;
END; ELSE FLAG
THEN DO;
IF AT T1(I2) = 0
THEN CALL EMIT('POINTR(' || S || ')=0;');
ELSE CALL_TABLESET(T,S);
ENTER_FLAG = FALSE;
END; ELSE FLAG
IF SET_FLAG
THEN DO;
TABLE_OP(1) || SUBSTR(ST2,LP,1);
TABLE_OP(0) || SUBSTR(ST2,RP,1);
T = T || SUBSTR(ST2,SEMI,1);
T = T || EMIT(T);
CALL_FLAG = FALSE;
SET_FLAG = 0;
TABLE_PTR = 0;

```



```

END; CALL EMIT(SUBSTR(ST2,SEMI,1));
ELSE PART = FALSE;
CODING = FALSE;
LOC_FLAG = FALSE;
END;

/* CASE 57 */
/* <ASSIGNMENT> ::= <LEFT PART> <BOOLEAN> */
DO; S = VAR(MP);
T = VAR(SP);
I1 = LOOKUP(S);
IF TYPE(I1) = CELL_ID & TYPE(I1) = FLAG_ID
THEN CALL ERROR('INVALID LEFT PART',1);
ELSE DO;
IF T = SUBSTR(ST3,TRU+1,4)
THEN CHECK = TRUE; ELSE
IF T = SUBSTR(ST3,FALS+1,5)
THEN CHECK = TRUE; ELSE
IF TEST_FLAG
THEN CHECK = TRUE;
IF CHECK
THEN CALL EMIT(SUBSTR(ST2,SEMI,1));
ELSE CALL ERROR('INVALID RIGHT PART',1);
CHECK = FALSE;
END;
TEST_FLAG = FALSE;
LEFT_PART = FALSE;
END;

/* CASE 58 */
/* <LEFT PART> ::= <PRIMARY> := */
DO; IF SET_FLAG
THEN DO;
LEFT DO; SET = TRUE;
SET_FLAG = FALSE;
END; ELSE
IF LIST_PTR > 0
THEN LIST_FLAG = TRUE; ELSE
IF CODING
THEN DO;
S = VAR(MP);
I1 = LOOKUP(S);
IF I1 = C

```



```

THEN CALL ERRORUN; ELSE
IF TYPE(I1) = STACK_ID
THEN PUSH = TRUE;
ELSE CALL EMIT(S || SUBSTR(ST3,EQ+1,1));
END;
LOCODE = VAR(MP);
LEFT_PART = TRUE;
END;

/* CASE 59 */
<OPERATION> ::= <TABLE OPERATION> */
DO;
END;

/* CASE 60 */
<OPERATION> ::= <STACK OPERATION> */
DO;
END;

/* CASE 61 */
<OPERATION> ::= <CODE OPERATION> */
DO;
END;

/* CASE 62 */
<OPERATION> ::= <AUXILIARY OPERATION> */
DO;
END;

/* CASE 63 */
<TABLE OPERATION> ::= <DELETE OPERATION> */
DO;
END;

/* CASE 64 */
<TABLE OPERATION> ::= <ENTER OPERATION> */
DO;
ENTER_FLAG = FALSE;
END;

/* CASE 65 */
<DELETE OPERATION> ::= <DELETE> <VARIABLE> ) */

```



```

DO: U = VAR(MPPI);
  IF SUBSTR(U,0,4) = VR | SUBSTR(U,0,5) = FV
  THEN T = U;
  ELSE DO:
    T = T || SUBSTR(ST2,QUOT,1) || U;
    T = T || SUBSTR(ST2,QUOT,1);
  END;
  T = T || SUBSTR(ST2,COMMA,1);
  T = T || VAR(MP) || SUBSTR(ST2,RP,1);
  T = T || SUBSTR(ST2,SEMI,1);
  CALL EMIT(T);
END;

/* CASE 66 */
/* <DELETE OPERATION> ::= DELETE <IDENTIFIER> */

DO: S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERROR; ELSE
  IF TYPE(I1) != CELL ID
  THEN CALL ERROR('INVALID DELETE OPERAND',1);
  ELSE DO:
    T = SUBSTR(ST4,DELX,13) || S;
    T = T || SUBSTR(ST2,RP,1);
    T = T || SUBSTR(ST2,SEMI,1);
    CALL EMIT(T);
  END;
END;

/* CASE 67 */
/* <DELETE> ::= DELETE ( <IDENTIFIER> */

DO: S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERROR; ELSE
  IF TYPE(I1) != TABLE ID
  THEN CALL ERROR('INVALID DELETE OPERAND',1);
  ELSE DO:
    CALL EMIT(SUBSTR(ST2,DEL,12));
    VAR(MP) = S;
  END;
END;

```



```

/* CASE 68 */
<VARIABLE> ::= <PRIMARY> */

DO; VAR(MP) = VAR(SP);
END;

/* CASE 69 */
<ENTER OPERATION> ::= <ENTRY POINT1> <LIST OPERAND> ) */

DO; S = TABLE_OP(0);
    I1 = LOOKUP(VAR(MP));
    DO I2 = 1 TO TABLE_PTR;
        I3 = I1 + I2;
        IF TYPE(I3) = TABLE_ENTRY
            THEN CALL ERROR('WRONG NUMBER OF ENTER PARAMETERS',1);
        ELSE DO;
            T = SYMBOL(I3) || SUBSTR(ST2,LP,1);
            T = T || SUBSTR(ST2,RP,1);
            T = T || SUBSTR(ST3,EQ+1,1);
            T = T || TABLE_OP(I2) || SUBSTR(ST2,SEMI,1);
            CALL EMIT(T);
        END;
    END;
    TABLE_PTR = 0;
END;

/* CASE 70 */
<ENTER OPERATION> ::= <ENTRY POINT1> <PRIMARY> ) */

DO; S = TABLE_OP(0);
    I1 = LOOKUP(VAR(MP)) + 1;
    IF TYPE(I1) = TABLE_ENTRY
        THEN CALL ERROR('INVALID ENTER PARAMETER',1);
    ELSE DO;
        T = SYMBOL(I1) || SUBSTR(ST2,LP,1);
        T = T || SUBSTR(ST2,RP,1);
        T = T || SUBSTR(ST3,EQ+1,1);
        T = T || VAR(MPPI) || SUBSTR(ST2,SEMI,1);
        CALL EMIT(T);
    END;
END;

/* CASE 71 */
<ENTRY POINT1> ::= <ENTRY POINT> */

```



```

DO;
END;

/* CASE 72 */
/* <ENTRY POINT> ::= <ENTER> <VARIABLE> : */

DO;
  S = VAR(MP);
  U = VAR(MPPI);
  I1 = LOOKUP(S);
  IF I1 = 0 THEN CALL ERROR; ELSE
  IF TYPE(I1) = TABLE_ID THEN CALL ERROR('INVALID ENTER OPERAND',1);
  ELSE DO;
    IF SUBSTR(U,0,4) = VR | SUBSTR(U,0,5) = FV
    THEN T = U;
    ELSE DO;
      T = SUBSTR(ST2,QUOT,1) || U;
      T = T || SUBSTR(ST2,QUOT,1);
    END;
    T = T || SUBSTR(ST2,COMMA,1) || S;
    T = T || SUBSTR(ST2,RP,1);
    T = T || SUBSTR(ST2,SEMI,1);
    CALL EMIT(T);
    IF AT1(I1) = 0 THEN CALL EMIT('POINTR(I1)=0;I2=I1;');
    ELSE CALL TABLESET(S,'I1');
    TABLE_OP(TABLE_PTR) = 'I2';
    TABLE_PTR = TABLE_PTR + 1;
  END;
END;

/* CASE 73 */
/* <ENTER> ::= ENTER ( <IDENTIFIER> */

DO;
  ENTER_FLAG = TRUE;
  VAR(MP) = VAR(SP);
  T = SUBSTR(ST4,AI1,2) || SUBSTR(ST3,EQ+1,1);
  T = T || SUBSTR(ST2,NTR,6);
  CALL EMIT(T);
END;

/* CASE 74 */
/* <STACK OPERATION> ::= POP <IDENTIFIER> */

```



```

DO; S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERROR; ELSE
  IF TYPE(I1) = STACK_ID
  THEN CALL ERROR('INVALID POP OPERAND',1);
  ELSE DO;
    CALL EMIT('IF ' || S || ' TOP = 0 THEN ');
    CALL EMIT('CALL ERROR(' || STACK_EMPTY || '); ELSE DO;');
    CALL EMIT('CALL POP(' || S || ' TOP);');
    CALL EMIT(S || ' _TOP=STKOV(' || S || ' _TOP);');
    IF AT(I1) = 0
    THEN CALL EMIT('PTR=' || S || ' _PTR-1;');
    CALL EMIT(SUBSTR(S12,FIN,4));
  END;
END;

/* CASE 75 */
/* <CODE OPERATION> ::= NOCODE */

DO;
END;

/* CASE 76 */
/* <CODE OPERATION> ::= <CODE> */

DO;
END;

/* CASE 77 */
/* <CODE> ::= <DIRECT> */

DO;
END;

/* CASE 78 */
/* <CODE> ::= <INDIRECT> */

DO;
END;

/* CASE 79 */
/* <DIRECT> ::= CODE <CHARACTER STRING> */

DO; CALL EMIT(VAR(SP));
END;

```



```

/* CASE 80 */
/* <INDIRECT> ::= <CODE 1> <CHARACTER STRING> ) */
DO;
  S = VAR(MPP1);
  IF CHECKCODE(S)
  THEN DO;
    T = SUBSTR(ST5,MIT,10) || S;
    T = T || SUBSTR(ST2,RP,1);
    T = T || SUBSTR(ST2,SEMI,i);
    CALL EMIT(T);
  END;
  ELSE CALL ERROR('INVALID CODE STREAM',1);
END;

/* CASE 81 */
/* <INDIRECT> ::= <CODE 1> <OPERAND PAIR> ) */
DO;
  END;

/* CASE 82 */
/* <INDIRECT> ::= <CODE 1> <CONTENTS> ) */
DO;
  T = SUBSTR(ST5,MIT,10) || VAR(MPP1);
  T = T || SUBSTR(ST2,RP,1);
  T = T || SUBSTR(ST2,SEMI,i);
  CALL EMIT(T);
  T = SUBSTR(ST5,MITX,11) || SUBSTR(ST5,LOD,12);
  CALL EMIT(T);
END;

/* CASE 83 */
/* <CODE 1> ::= CODE ( */
DO;
  END;

/* CASE 84 */
/* <OPERAND PAIR> ::= <OPERATOR> */
DO;
  END;

/* CASE 85 */
/* <OPERAND PAIR> ::= <OPERATOR> , <CHARACTER STRING> */

```



```

DO; S = VAR(SP);
  IF CHECKCODE(S)
  THEN DO;
    T = SUBSTR(ST5,MIT,10) || S;
    T = T || SUBSTR(ST2,RP,1);
    T = T || SUBSTR(ST2,SEMI,1);
    CALL EMIT(T);
  END;
  ELSE CALL ERROR('INVALID CODE STREAM',1);
END;

/* CASE 86 */
/* <OPERATOR> ::= OP <CHARACTER STRING> */

DO; T = SUBSTR(ST5,MITX,11) || VAR(SP);
  T = T || SUBSTR(ST3,ORO,3);
  T = T || SUBSTR(ST5,EIGHTY,4);
  T = T || SUBSTR(ST2,RP,1);
  T = T || SUBSTR(ST2,SEMI,1);
  CALL EMIT(T);
END;

/* CASE 87 */
/* <CONTENTS> ::= CONT <CHARACTER STRING> */

DO; S = VAR(SP);
  IF CHECKCODE(S)
  THEN VAR(MP) = S;
  ELSE CALL ERROR('INVALID CODE STREAM',1);
END;

/* CASE 88 */
/* <AUXILIARY OPERATION> ::= TALLY <IDENTIFIER> */

DO; S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERRORUN;
  ELSE DO;
    IF TYPE(I1) ^= CELL_ID
    THEN CALL ERROR('INVALID TALLY OPERAND',1);
    ELSE DO;
      T = S || SUBSTR(ST3,EQ+1,1);
    END;
  END;

```



```

T = T || S || SUBSTR(ST4,P1,2);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
END;
END;

/* CASE 89 */
<AUXILIARY OPERATION> ::= MINUS <IDENTIFIER> */

DO; S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERRORUN;
  ELSE DO;
    IF TYPE(I1) = CELL_ID
    THEN CALL ERROR('INVALID MINUS OPERAND',1);
    ELSE DO;
      T = S || SUBSTR(ST3,EQ+1,1);
      T = T || S || SUBSTR(ST4,M1,2);
      T = T || SUBSTR(ST2,SEMI,1);
      CALL EMIT(T);
    END;
  END;
END;

/* CASE 90 */
<AUXILIARY OPERATION> ::= <ERROR> <PRIMARY> */

DO; T = SUBSTR(ST2,QUOT,1) || VAR(SP);
  T = T || SUBSTR(ST2,QUOT,1);
  T = T || SUBSTR(ST2,RP,1);
  T = T || SUBSTR(ST2,SEMI,1);
  CALL EMIT(T);
END;

/* CASE 91 */
<AUXILIARY OPERATION> ::= STOP */

DO; CALL EMIT(SUBSTR(ST2,XIT,10));
END;

/* CASE 92 */
<ERROR> ::= ERROR */

```



```

DO; CALL EMIT(SUBSTR(ST2,ERR,11));
END;

/* CASE 93 */
<TRANSFER> ::= JUMP <IDENTIFIER> */

DO; S = VAR(SP);
    I1 = LOOKUP(S);
    IF I1 = 0
    THEN CALL ERROR('UNDECLARED LABEL',1);
    ELSE DO;
        IF TYPE(I1) != TYPE_LABEL
        THEN CALL ERROR('INVALID JUMP OPERAND',1);
        ELSE DO;
            T = SUBSTR(ST2,GOTU,5) || S;
            T = T || SUBSTR(ST2,SEMI,1);
            CALL EMIT(T);
        END;
    END;
END;

/* CASE 94 */
<CONDITIONAL> ::= <IF CLAUSE> <THEN CLAUSE> */

DO;
END;

/* CASE 95 */
<CONDITIONAL> ::= <IF CLAUSE> <THEN CLAUSE> <ELSE CLAUSE> */

DO;
END;

/* CASE 96 */
<IF CLAUSE> ::= <IF> <BOOLEAN> */

DO; BOOL = FALSE;
END;

/* CASE 97 */
<IF> ::= IF */

DO; BOOL = TRUE;
    CALL EMIT(SUBSTR(ST2,FI,3));

```



```

END;

/* CASE 98 */
<THEN CLAUSE> ::= <THENDO> <SENTENCE> END ; */

DO;
CALL EMIT(SUBSTR(ST2,FIN,4));
END;

/* CASE 99 */
<THENDO> ::= THENDO ; */

DC;
CALL EMIT(SUBSTR(ST2,TD,9));
END;

/* CASE 100 */
<ELSE CLAUSE> ::= <ELSEDO> <SENTENCE> END ; */

DO;
CALL EMIT(SUBSTR(ST2,FIN,4));
END;

/* CASE 101 */
<ELSEDO> ::= ELSEDO ; */

DO;
CALL EMIT(SUBSTR(ST2,ED,8));
END;

/* CASE 102 */
<BOOLEAN> ::= <BOOLEAN TERM> */

DO;
END;

/* CASE 103 */
<BOOLEAN> ::= <BOOLEAN> <OR> <BOOLEAN TERM> */

DO;
END;

/* CASE 104 */
<OR> ::= OR */

DO;
CALL EMIT(SUBSTR(ST3,ORO,3));
END;

```



```

/* CASE 105 */ <BOOLEAN TERM> ::= <BOOLEAN FACTOR> */
/*
DC;
END;

/* CASE 106 */
/* <BOOLEAN TERM> ::= <BOOLEAN TERM> <AND> <BOOLEAN FACTOR> */
/*
DO;
END;

/* CASE 107 */
/* <AND> ::= AND */
/*
DC;
CALL EMIT(SUBSTR(ST3,AND0,3));
END;

/* CASE 108 */
/* <BOOLEAN FACTOR> ::= <BOOLEAN PRIMARY> */
/*
DO;
END;

/* CASE 109 */
/* <BOOLEAN FACTOR> ::= <NOT> <BOOLEAN PRIMARY> */
/*
DO;
END;

/* CASE 110 */
/* <NOT> ::= NOT */
/*
DO; NOT_FLAG = TRUE;
END;

/* CASE 111 */
/* <BOOLEAN PRIMARY> ::= <ARITHMETIC> <RELATION> <ARITHMETIC> */
/*
DO;
END;

/* CASE 112 */
/* <BOOLEAN PRIMARY> ::= <LDEL*> <BOOLEAN> <RDEL*> */

```



```

DO;
END;

/* CASE 113 */
/* <BCLEAN PRIMARY> ::= <TEST> <PRIMARY> */

DO; S = VAR(SP);
  I1 = LOOKUP(S);
  IF TYPE(I1) = STACK_ID
  THEN DO;
    IF LEFT PART
    THEN DO;
      T = SUBSTR(ST3,TRU+1,4) || SUBSTR(ST2,SEMI,1);
      CALL EMIT(T);
      T = SUBSTR(ST2,FI,3) || S;
      T = T || SUBSTR(ST2,PTR,4);
      IF NOT FLAG
      THEN T = T || SUBSTR(ST3,GT,3);
      ELSE T = T || SUBSTR(ST3,LE,4);
      T = T || SUBSTR(ST5,ZERO,1);
      CALL EMIT(T);
      T = SUBSTR(ST2,TD,6) || LOCORE;
      T = T || SUBSTR(ST3,EQ+1,1);
      T = T || SUBSTR(ST3,FALS+1,5);
      CALL EMIT(T);
    END; ELSE
    IF BOOL
    THEN DO;
      T = S || SUBSTR(ST2,PTR,4);
      IF NOT FLAG
      THEN T = T || SUBSTR(ST3,LE,4);
      ELSE T = T || SUBSTR(ST3,GT,3);
      T = T || SUBSTR(ST5,ZERO,1);
      CALL EMIT(T);
    END; ELSE
    END; ELSE
    IF TYPE(I1) = CELL_ID
    THEN DO;
      IF LEFT PART
      THEN DO;
        T = SUBSTR(ST3,TRU+1,4) || SUBSTR(ST2,SEMI,1);
        CALL EMIT(T);
        T = SUBSTR(ST2,FI,3) || S;
        IF NOT FLAG
        THEN T = T || SUBSTR(ST3,GT,3);
        ELSE T = T || SUBSTR(ST3,LE,4);
        T = T || SUBSTR(ST5,ZERO,1);
      END;
    END;
  END;

```



```

CALL EMIT(T);
T = SUBSTR(ST2,TD,6) || LOC CODE;
T = T || SUBSTR(ST3,EQ+1,1);
T = T || SUBSTR(ST3,FALS+1,5);
CALL EMIT(T);
END; ELSE
IF BOOL
THEN DO;
IF NOT FLAG
THEN T = S || SUBSTR(ST3,LE,4);
ELSE T = S || SUBSTR(ST3,GT,3);
T = T || SUBSTR(ST5,ZERO,1);
CALL EMIT(T);
END;
END; ELSE
IF TYPE(I1) = STRING_ID
THEN DO;
IF LEFT PART
THEN T = SUBSTR(ST3,TRU+1,4) || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST2,FI,3) || S;
IF NOT FLAG
THEN T = T || SUBSTR(ST3,EQ,3);
ELSE T = T || SUBSTR(ST3,NE,4);
T = T || SUBSTR(ST5,NIL,3);
CALL EMIT(T);
T = SUBSTR(ST2,TD,6) || LOC CODE;
T = T || SUBSTR(ST3,EQ+1,1);
T = T || SUBSTR(ST3,FALS+1,5);
CALL EMIT(T);
END; ELSE
IF BOOL
THEN DO;
IF NOT FLAG
THEN T = S || SUBSTR(ST3,NE,4);
ELSE T = S || SUBSTR(ST3,EQ,3);
T = T || SUBSTR(ST5,NIL,3);
CALL EMIT(T);
END;
END; ELSE
IF TYPE(I1) = FLAG_ID
THEN CALL EMIT(S);
ELSE CALL ERROR('INVALID TEST OPERAND',1);
TEST FLAG = TRUE;
NOT FLAG = FALSE;
END;

```



```

/* CASE 114 */
/* <BOOLEAN PRIMARY> ::= TRUE */
DO;
IF NOT_FLAG
THEN DO;
CALL_EMIT(SUBSTR(ST3,FALS+1,5));
NOT_FLAG = FALSE;
END;
ELSE CALL_EMIT(SUBSTR(ST3,TRU+1,4));
END;

/* CASE 115 */
/* <BOOLEAN PRIMARY> ::= FALSE */
DO;
IF NOT_FLAG
THEN DO;
CALL_EMIT(SUBSTR(ST3,TRU+1,4));
NOT_FLAG = FALSE;
END;
ELSE CALL_EMIT(SUBSTR(ST3,FALS+1,5));
END;

/* CASE 116 */
/* <TEST> ::= TEST */
DO;
END;

/* CASE 117 */
/* <RELATION> ::= .LSS. */
DO;
IF NOT_FLAG
THEN DO;
CALL_EMIT(SUBSTR(ST3,GE,4));
NOT_FLAG = FALSE;
END;
ELSE CALL_EMIT(SUBSTR(ST3,LT,3));
END;

/* CASE 118 */
/* <RELATION> ::= .LEQ. */
DO;
IF NOT_FLAG
THEN DO;

```



```

CALL EMIT(SUBSTR(ST3,GT,3));
NOT_FLAG = FALSE;
END;
ELSE CALL EMIT(SUBSTR(ST3,LE,4));
END;

/* CASE 119 */
/* <RELATION> ::= .EQL. */
DO;
IF NOT_FLAG
THEN DO;
CALL EMIT(SUBSTR(ST3,NE,4));
NOT_FLAG = FALSE;
END;
ELSE CALL EMIT(SUBSTR(ST3,EQ,3));
END;

/* CASE 120 */
/* <RELATION> ::= .NEQ. */
DO;
IF NOT_FLAG
THEN DO;
CALL EMIT(SUBSTR(ST3,EQ,3));
NOT_FLAG = FALSE;
END;
ELSE CALL EMIT(SUBSTR(ST3,NE,4));
END;

/* CASE 121 */
/* <RELATION> ::= .GEQ. */
DO;
IF NOT_FLAG
THEN DO;
CALL EMIT(SUBSTR(ST3,LT,3));
NOT_FLAG = FALSE;
END;
ELSE CALL EMIT(SUBSTR(ST3,GE,4));
END;

/* CASE 122 */
/* <RELATION> ::= .GTR. */
DO;
IF NOT_FLAG
THEN DO;

```



```

CALL EMIT(SUBSTR(ST3,LE,4));
NOT_FLAG = FALSE;
END;
ELSE CALL EMIT(SUBSTR(ST3,GT,3));
END;

/* CASE 123 */
<ARITHMETIC> ::= <TERM> */

DO;
END;

/* CASE 124 */
<ARITHMETIC> ::= <+> <TERM> */

DO;
END;

/* CASE 125 */
<ARITHMETIC> ::= <-> <TERM> */

DO;
END;

/* CASE 126 */
<ARITHMETIC> ::= <ARITHMETIC> <+> <TERM> */

DO;
END;

/* CASE 127 */
<ARITHMETIC> ::= <ARITHMETIC> <-> <TERM> */

DO;
END;

/* CASE 128 */
<+> ::= + */

DO;
CALL EMIT(SUBSTR(ST3,PLUS,1));
END;

/* CASE 129 */
<-> ::= - */

DO;
CALL EMIT(SUBSTR(ST3,MINUS,1));

```



```

END;

/* CASE 130 */
<TERM> ::= <FACTOR> */

DO;
END;

/* CASE 131 */
<TERM> ::= <TERM> <*> <FACTOR> */

DO;
END;

/* CASE 132 */
<TERM> ::= <TERM> </> <FACTOR> */

DO;
END;

/* CASE 133 */
<*> ::= * */

DO; CALL EMIT(SUBSTR(ST3, TIMES, 1));
END;

/* CASE 134 */
</> ::= / */

DO; CALL EMIT(SUBSTR(ST3, DIVI, 1));
END;

/* CASE 135 */
<FACTOR> ::= <PRIMARY> */

DO;
  IF LEFT_PART | BOOL THEN
  IF ~SET_FLAG & ~ENTER_FLAG THEN
  IF ~TOP_FLAG & ~PUSH THEN
  IF ~LIST_FLAG & ~LOC_FLAG THEN
  THEN DO;
    T = SUBSTR(ST2, QUOT, 1) || VAR(MP);
    T = T || SUBSTR(ST2, QUOT, 1);
    CALL EMIT(T);
    CHAR_FLAG = FALSE;
  
```



```

END;
ELSE CALL EMIT(VAR(MP));
END;

/* CASE 136 */
/* <FACTOR> ::= <FACTOR> <*> <PRIMARY> */

DO; CALL EMIT(VAR(SP));
END;

/* CASE 137 */
/* <*> ::= ** */

DO; CALL EMIT(SUBSTR(ST3, EXPO, 2));
END;

/* CASE 138 */
/* <PRIMARY> ::= <OPERAND> */

DO; CHAR_FLAG = FALSE;
END;

/* CASE 139 */
/* <PRIMARY> ::= <NUMBER> */

DO; VAR(MP) = NUMBER_VALUE;
END;

/* CASE 140 */
/* <PRIMARY> ::= <BIT STRING> */

DO;
END;

/* CASE 141 */
/* <PRIMARY> ::= <CHARACTER STRING> */

DO; CHAR_FLAG = TRUE;
END;

/* CASE 142 */
/* <PRIMARY> ::= <LDEL*> <ARITHMETIC> <RDEL*> */

```



```

DO;
END;

/* CASE 143 */
<OPERAND> ::= <STABLE OPERAND> */

DO;
END;

/* CASE 144 */
<OPERAND> ::= <FLOATING OPERAND> */

DO;
END;

/* CASE 145 */
<OPERAND> ::= <STORAGE OPERAND> */

DO;
END;

/* CASE 146 */
<OPERAND> ::= <CODE OPERAND> */

DO;
END;

/* CASE 147 */
<OPERAND> ::= ( <LIST OPERAND> ) */

DO;
VAR(MP) = NUL;
END;

/* CASE 148 */
<STABLE OPERAND> ::= <ENTRY OPERAND> */

DO;
END;

/* CASE 149 */
<STABLE OPERAND> ::= <LOC OPERAND> */

DO;
END;

/* CASE 150 */
<STABLE OPERAND> ::= TOP <IDENTIFIER> */

```



```

DO; TOP_FLAG = TRUE;
  VAR(MP) = VAR(SP);
END;

/* CASE 151 */
/* <STABLE OPERAND> ::= <SET OPERAND> */

DO;
END;

/* CASE 152 */
/* <LOC OPERAND> ::= <LOC> <VARIABLE> */

DO;
  S = VAR(MP);
  U = VAR(SP);
  IF SUBSTR(U,0,4) = VR | SUBSTR(U,0,5) = FV.
  THEN T = U;
  ELSE DO;
    T = T || SUBSTR(ST2,QUOT,1) || U;
    T = T || SUBSTR(ST2,QUOT,1);
  END;
  T = T || SUBSTR(ST2,COMMA,1) || S;
  T = T || SUBSTR(ST2,RP,1);
  CALL EMIT(T);
  I1 = LOOKUP(S);
  IF ATT1(I1) = 0
  THEN DO;
    T = SUBSTR(ST2,SEMI,1) || LOCODE;
    T = T || SUBSTR(ST3,EQ+1,1);
    T = T || SUBSTR(ST4,POINT,7);
    T = T || LOCODE || SUBSTR(ST2,RP,1);
    CALL EMIT(T);
  END;
  VAR(MP) = NUL;
  LOC_FLAG = TRUE;
END;

/* CASE 153 */
/* <LOC> ::= LOC ( <IDENTIFIER> */

DO;
  S = VAR(SP);
  VAR(MP) = S;
  I1 = LOOKUP(S);
  IF I1 = 0

```



```

THEN CALL ERRORUN; ELSE
IF TYPE(I1) /= TABLE_ID
THEN CALL ERROR('INVALID LOC OPERAND',1);
ELSE CALL EMIT(SUBSTR(ST4,LOOK,7));
END;

/* CASE 154 */
/* <ENTRY OPERAND> ::= <ENTRY> <VARIABLE> */

DO;
S = VAR(MP);
U = VAR(SP);
IF SUBSTR(U,0,4) = VR | SUBSTR(U,0,5) = FV
THEN T = U;
ELSE DO;
T = SUBSTR(ST2,QUOT,1) || U;
T = T || SUBSTR(ST2,QUOT,1);
END;
T = T || SUBSTR(ST2,COMMA,1) || S;
T = T || SUBSTR(ST2,RP,1);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
END;

/* CASE 155 */
/* <ENTRY> ( <IDENTIFIER> */

DO;
S = VAR(SP);
VAR(MP) = S;
I1 = LOOKUP(S);
IF I1 = C
THEN CALL ERRORUN; ELSE
IF TYPE(I1) /= TABLE_ID
THEN CALL ERROR('INVALID ENTRY OPERAND',1);
ELSE DO;
ENTER_FLAG = TRUE;
CALL EMIT(SUBSTR(ST2,NTR,6));
END;
END;

/* CASE 156 */
/* <SET OPERAND> ::= <SET POINT> <LIST OPERAND> */

DO;
END;

/* CASE 157 */

```



```

/* <SET OPERAND> ::= <SET POINT> <PRIMARY> */
DO;
  S = VAR(SP);
  I1 = LOOKUP(S);
  IF I1 = 0
  THEN CALL ERROR; ELSE
  IF TYPE(I1) = TABLE_ENTRY & TYPE(I1) = STACK_ENTRY
  THEN CALL ERROR('INVALID SET OPERAND',1);
  ELSE DO;
    TABLE_OP(TABLE_PTR) = S;
  END;
END;

/* CASE 158 */
/* <SET POINT> ::= <SET POINT 1> <VARIABLE> */
IF LEFT_PART
THEN DO;
  S = VAR(MP);
  U = VAR(MPPI);
  I1 = LOOKUP(S);
  IF TYPE(I1) = TABLE_ID
  THEN DO;
    T = SUBSTR(ST4,A1,2) || SUBSTR(ST3,EQ+1,1);
    IF SUBSTR(U,C,4) = VR || SUBSTR(U,C,5) = FV
    THEN SUBSTR(T,1,1) = U;
    ELSE DO;
      T = SUBSTR(ST2,QUOT,1) || U;
    END;
  CALL EMIT(T);
  CALL SUBSTR(ST2,COMMA,1) || S;
  T = T || SUBSTR(ST2,RP,1);
  T = T || SUBSTR(ST2,SEMI,1);
  CALL EMIT(T);
  IF ATT1(I1) = 0
  THEN DO;
    T = SUBSTR(ST4,POINT,7) || SUBSTR(ST4,A1,2);
    T = T || SUBSTR(ST2,RP,1);
    T = T || SUBSTR(ST4,EZ,3);
    T = T || SUBSTR(ST2,SEMI,1);
    CALL EMIT(T);
    CALL EMIT(SUBSTR(ST4,I2,1,6));
  END;
  DO;
    ELSE T = SUBSTR(ST2,FI,3) || S;
  END;

```



```

T = T || SUBSTR(ST2,PTR,4);
T = T || SUBSTR(ST3,GT,3) || S;
T = T || SUBSTR(ST2,EX,3);
CALL EMIT(T);
T = SUBSTR(ST2,TD,9) || 'CALL ERROR('TABLE OVERFLOW')';
CALL EMIT(T);
T = SUBSTR(ST2,XIT,10) || SUBSTR(ST2,FIN,4);
CALL EMIT(T);
T = SUBSTR(ST4,I2I1,3) || S;
T = T || SUBSTR(ST2,PTR,4);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST4,POINT,7) || SUBSTR(ST4,I1I2,7);
CALL EMIT(T);
T = S || SUBSTR(ST2,PTR,4);
T = T || SUBSTR(ST3,EQ+1,1) || S;
T = T || SUBSTR(ST2,PTR,4);
T = T || SUBSTR(ST4,P1,2);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
END; ELSE
IF TYPE(I1) = STACK_ID
THEN DO:
T = SUBSTR(ST4,AI1,2) || SUBSTR(ST3,EQ+1,1);
T = T || SUBSTR(ST4,STK,6);
IF SUBSTR(U,0,4) = VR || SUBSTR(U,0,5) = FV
THEN T = T || U;
ELSE DO:
T = T || SUBSTR(ST2,QUOT,1) || U;
T = T || SUBSTR(ST2,QUOT,1);
END;
CALL EMIT(T);
T = SUBSTR(ST2,COMMA,1) || S;
T = T || SUBSTR(ST2,RP,1);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST4,SQVR,11) || S;
T = T || SUBSTR(ST2,TOP,4);
T = T || SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST2,TOP,4);
T = T || SUBSTR(ST4,I2I1+2,4);
CALL EMIT(T);
IF ATT(I1) = 0
THEN DO:
T = SUBSTR(ST4,SPTR,7) || SUBSTR(ST4,I1I2,3);
T = T || SUBSTR(ST4,EZ,3);

```



```

CALL EMIT(T);
CALL EMIT(SUBSTR(ST4,I2I1,6));
END;
ELSE
DO;
SUBSTR(ST2,FI,3) || S;
SUBSTR(ST2,PTR,4);
SUBSTR(ST3,GT,3) || S;
SUBSTR(ST2,EX,3);
CALL EMIT(T);
T = SUBSTR(ST2,TD,9) || 'CALL ERROR(''STACK OVERFLOW'')';
CALL EMIT(T);
T = SUBSTR(ST2,XIT,10) || SUBSTR(ST2,FIN,4);
CALL EMIT(T);
SUBSTR(ST4,I2I1,3) || S;
SUBSTR(ST2,PTR,4);
SUBSTR(ST2,SEMI,1);
CALL EMIT(T);
T = SUBSTR(ST4,SPTR,7) || SUBSTR(ST4,I1I2,7);
CALL EMIT(T);
SUBSTR(ST2,PTR,4);
SUBSTR(ST3,EQ+1,1) || S;
SUBSTR(ST2,PI,2);
SUBSTR(ST2,SEMI,j);
CALL EMIT(T);
END;
END;
ELSE CALL ERROR('INVALID USE OF SET OPERATOR',1);

/* CASE 159 */
<SET POINT> ::= <SET POINT 1> :      */

DO;
IF TYPE(I1) = STACK_ID
THEN TABLE_OP(2) = S || '_PTR-1';
END;

/* CASE 160 */
<SET POINT 1> ::= <SET> <IDENTIFIER>      */

DO;
S = VAR(SP);
I1 = LOOKUP(S);
IF I1 = 0
THEN CALL ERRORUN: ELSE
IF TYPE(I1) = CELL_ID
THEN DO;

```



```

TABLE_OP(TABLE_PTR) = S;
TABLE_PTR = TABLE_PTR + 1;
END; ELSE
IF TYPE(I1) = TABLE_ID & TYPE(I1) = STACK_ID
THEN CALL ERROR('INVALID SET OPERAND',1);
ELSE DO;
TABLE_OP(TABLE_PTR) = SUBSTR(ST4,A12,2);
TABLE_PTR = TABLE_PTR + 1;
END;
VAR(MP) = S;
END;

/* CASE 161 */
/* <SET> ::= SET ( */
DO; SET_FLAG = TRUE;
END;

/* CASE 162 */
/* <FLOATING OPERAND> ::= <FLOP> <PRIMARY> ) */
DO; S = VAR(MPPI);
IF LEFT_PART
THEN DO;
T = SUBSTR(ST5,RESTL,16);
IF SUBSTR(S,0,4) = VR | SUBSTR(S,0,5) = FV
THEN T = T || S;
ELSE DO;
T = T || SUBSTR(ST2,QUOT,1) || S;
END;
T = T || SUBSTR(ST2,RP,1);
CALL EMIT(T);
END;
ELSE FLOP = S;
VAR(MP) = NUL;
LOC_FLAG = TRUE;
END;

/* CASE 163 */
/* <FLOATING OPERAND> ::= <FLOP> <PROGRAM LABEL> ) */
DO; S = VAR(MPPI);
IF LEFT_PART
THEN DO;

```



```

T = SUBSTR(ST6,RESTA,18);
IF SUBSTR(S,0,4) = VR | SUBSTR(S,0,5) = FV
THEN T = T || S;
ELSE DO;
  T = T || SUBSTR(ST2,QUOT,1) || S;
  T = T || SUBSTR(ST2,QUOT,1);
END;
T = T || SUBSTR(ST2,RP,1);
CALL EMIT(T);
END;
ELSE FLOP = S;
VAR(MP) = NUL;
LOC_FLAG = TRUE;
END;

/* CASE 164 */
<FLOP> ::= FLOP (
DO; CODING = TRUE;
END;

/* CASE 165 */
<PROGRAM LABEL> ::= ADDR <PRIMARY>
DO; ADD_FLAG = TRUE;
  VAR(MP) = VAR(SP);
END;

/* CASE 166 */
<STORAGE OPERAND> ::= <IDENTIFIER>
DO;
END;

/* CASE 167 */
<CCDE OPERAND> ::= CODELOC
DO; CODING = TRUE;
  IF LEFT_PART
  THEN DO;
    IF ADD_FLAG
    THEN DO;
      ADD_FLAG = FALSE;
      T = SUBSTR(ST6,SAVEA,15);
    END;
  END;

```



```

ELSE T = SUBSTR(ST5,SAVEL,13);
IF SUBSTR(FLOP,0,4) = VR | SUBSTR(FLOP,0,5) = FV
THEN T = T || FLOP;
ELSE DO;
  T = T || SUBSTR(ST2,QUOT,1) || FLOP;
  T = T || SUBSTR(ST2,QUOT,1);
END;
T = T || SUBSTR(ST2,RP,1);
CALL EMIT(T);
END;
VAR(MP) = NUL;
END;

/* CASE 168 */
/* <LIST OPERAND> ::= <PRIMARY> ; <PRIMARY> */

DO; S = VAR(MP);
  T = VAR(SP);
  IF SET_FLAG
  THEN DO;
    I1 = LOOKUP(S);
    I2 = LOOKUP(T);
    IF I1 = 0 | I2 = 0
    THEN CALL ERROR; ELSE
    IF TYPE(I1) /= TYPE(I2)
    THEN CALL ERROR('INVALID SET OPERAND(S)',1); ELSE
    IF TYPE(I1) /= TABLE_ENTRY &
    THEN CALL ERROR('INVALID SET OPERAND(S)',1);
    ELSE DO;
      TABLE_OP(TABLE_PTR) = S;
      TABLE_PTR = TABLE_PTR + 1;
      TABLE_OP(TABLE_PTR) = T;
    END; ELSE
    END; ELSE
    IF ENTER_FLAG
    THEN DO;
      TABLE_OP(TABLE_PTR) = S;
      TABLE_PTR = TABLE_PTR + 1;
      TABLE_OP(TABLE_PTR) = T;
    END; ELSE
    END; DO;
      IF LIST_FLAG
      THEN DO;
        LIST_LEVEL = LIST_PTR;
        LIST_PTR = LIST_PTR + 1;
        LIST_OP(LIST_PTR) = S;
        LIST_PTR = LIST_PTR + 1;
      END;
    END;
  END;

```



```

LIST_OP(LIST_PTR) = T;
END;
ELSE DO;
  LIST_OP(LIST_PTR) = S;
  LIST_PTR = LIST_PTR + 1;
  LIST_OP(LIST_PTR) = T;
END;
END;
END;

/* CASE 169 */
/* <LIST OPERAND> ::= <LIST OPERAND> ; <PRIMARY> */
DO;
  S = VAR(SP);
  IF SET_FLAG
  THEN DO;
    I1 = LOOKUP(S);
    IF I1 = 0
    THEN CALL ERROR;
    IF TYPE(I1) = TABLE_ENTRY & TYPE(I1) = STACK_ENTRY
    THEN CALL ERROR('INVALID SET OPERAND',1);
    ELSE DO;
      TABLE_PTR = TABLE_PTR + 1;
      IF TABLE_PTR > 16
      THEN DO;
        CALL ERROR('TOO MANY SET OPERANDS',1);
        TABLE_PTR = TABLE_PTR - 1;
      END;
      ELSE TABLE_OP(TABLE_PTR) = S;
    END;
  END;
  ELSE
  END;
  IF ENTER_FLAG
  THEN DO;
    TABLE_PTR = TABLE_PTR + 1;
    IF TABLE_PTR > 16
    THEN DO;
      CALL ERROR('TOO MANY ENTER PARAMETERS',1);
      TABLE_PTR = TABLE_PTR - 1;
    END;
    TABLE_OP(TABLE_PTR) = S;
  END;
  ELSE DO;
    LIST_PTR = LIST_PTR + 1;
    IF LIST_PTR > 31 | (-LIST_FLAG & LIST_PTR > 15)
    THEN DO;
      CALL ERROR('TOO MANY LIST PARAMETERS',1);
      LIST_PTR = LIST_PTR - 1;
    END;
  END;

```



```

END;
ELSE LIST_OP(LIST_PTR) = S;
END;
END;

/* CASE 170 */
<LDEL*> ::= < */

DO;
IF NOT_FLAG
THEN DO;
CALL EMIT(SUBSTR(ST3,NOTO,3));
NOT_FLAG = FALSE;
END;
CALL EMIT(SUBSTR(ST2,LP,1));
END;

/* CASE 171 */
<RDEL*> ::= > */

DO;
CALL EMIT(SUBSTR(ST2,RP,1));
END;

END;
END SYNTHESIZE;

```

```

/* */
/* */
/* */
/* */
SYNTACTIC PARSING FUNCTIONS
/* */
/* */
/* */
/* */

```

```

RIGHT_CONFLICT:
PROCEDURE(LEFT) BIT(1);
DECLARE LEFT FIXED;
RETURN ("CG" & SHL(BYTE(C1(LEFT),SHR(TOKEN,2)),SHL(TOKEN,1) & "06"))
= 0;
END RIGHT_CONFLICT;

```

```

RECOVER:
PROCEDURE;
IF FAILSOFT
THEN CALL SCAN;
FAILSOFT = FALSE;
DO WHILE NOT STOPIT(TOKEN);

```



```

CALL SCAN;
END; WHILE RIGHT_CONFLICT(PARSE_STACK(SP));
IF SP > 2
THEN SP = SP - 1;
ELSE CALL SCAN;
END;
OUTPUT = 'RESUME:' || SUBSTR(POINTER,TEXT_LIMIT-CP+MARGIN_CHOP+7);
END RECOVER;

STACKING:
PROCEDURE BIT(1);
CALLCOUNT(1) = CALLCOUNT(1) + 1;
DO FOREVER;
DO CASE SHR(BYTE(C1(PARSE_STACK(SP))),SHR(TOKEN,2)),SHL(3-TOKEN,1)
& 6) & 3;
/* CASE 0 - ILLEGAL SYMBOL PAIR */
DO; CALL ERROR('ILLEGAL SYMBOL PAIR; ' || V(PARSE_STACK(SP)) ||
X1 || V(TOKEN),1);
CALL STACK_DUMP;
CALL RECOVER;
END;
/* CASE 1 - STACK TOKEN */
RETURN TRUE;
/* CASE 2 - DON'T STACK TOKEN */
RETURN FALSE;
/* CASE 3 - CHECK TRIPLES */
DO; J = SHL(PARSE_STACK(SP-1),16) + SHL(PARSE_STACK(SP),8) +
TOKEN;
I = -1;
K = NCITRIPLES + 1;
DO WHILE I + 1 < K;
L = SHR(I+K,1);
IF CITRIPLES(L) > J
THEN K = L; ELSE
IF CITRIPLES(L) < J
THEN I = L;
ELSE RETURN TRUE;

```



```

END;
RETURN FALSE;
END;

END;
END STACKING;

PR_OK:
PROCEDURE (PRD) BIT(1);
DECLARE (H,I,J,PRD) FIXED;
DO CASE CONTEXT_CASE (PRD);
    /* CASE 0 - NO CHECK REQUIRED */
    RETURN TRUE;
    /* CASE 1 - RIGHT CONTEXT CHECK */
    RETURN ~ RIGHT_CONFLICT (HDTB (PRD));
    /* CASE 2 - LEFT CONTEXT CHECK */
DO;
    H = HDTB (PRD) - NT;
    I = PARSE_STACK (SP-PRLENGTH (PRD));
    DO J = LEFT_INDEX (H-1) TO LEFT_INDEX (H) - 1;
        IF LEFT_CONTEXT (J) = I
            THEN RETURN TRUE;
    END;
    RETURN FALSE;
END;
/* CASE 3 - CHECK TRIPLES */
DO;
    H = HDTB (PRD) - NT;
    I = SHL (PARSE_STACK (SP-PRLENGTH (PRD)), 8) + TOKEN;
    DO J = TRIPLE_INDEX (H-1) TO TRIPLE_INDEX (H) - 1;
        IF CONTEXT_TRIPLE (J) = I
            THEN RETURN TRUE;
    END;
    RETURN FALSE;
END;
END;
PR_OK;
END;

```


ANALYSIS ALGORITHM

CHECKING ABORTED ***', 2);

MISCELLANEOUS PROCEDURES

207

SUPERSKELETON

SUPER SKELETON

THE FOLLOWING CARDS ARE PUNCHED BY THE SYNTAX PRE-PROCESSOR

DECLARATIONS FOR THE SCANNER

DECLARE ALPHABET CHARACTER INITIAL


```

('ABCDEFGHIJKLMNPQRSTUVWXYZ_.$'),
BCD CHARACTER,
BUFFER CHARACTER,
CALLCOUNT(20) FIXED INITIAL(C,0,0,0,0),
C,C,0,0,C,C,0,0,0,0,0,0,0,0,
CARD COUNT FIXED,
CHARACTER TYPE(255) BIT(8),
CLOCK(5) FIXED,
CODELOC(FC_MAX) FIXED,
COLON FIXED,
COMPILING BIT(1),
CONTROL(255) BIT(1),
CP FIXED,
DIVIDE FIXED,
DOUBLE CHARACTER INITIAL ('O'),
EOF FILE FIXED,
EQS FIXED,
ERROR COUNT FIXED,
EXPON_FIXED,
FAILSOFT BIT(1),
FC_FREE FIXED INITIAL(1),
FC_PTR FIXED INITIAL(1),
FIXV(STACKSIZE) FIXED,
FLOPOVER(FC_MAX) CHARACTER,
FLOPOVER(FC_MAX) BIT(8),
FREEELIST FIXED INITIAL(1),
FREE_KEY FIXED INITIAL(1),
FREE_STK FIXED INITIAL(1),
HASH_TABLE(HASH_EX) BIT(16),
I1 FIXED,
I11 FIXED,
I12 FIXED,
I13 FIXED,
I14 FIXED,
IDENT FIXED,
ID(MAXKEY) BIT(8),
J FIXED,
K FIXED,
KEY(MAXKEY) CHARACTER,
L FIXED,
LABELADDR(MAXLABEL) FIXED,
LABELOVER(MAXLABEL) BIT(8),
LABELPTR FIXED INITIAL(1),
LABELS(MAXLABEL) CHARACTER,
LIST(MAXLIST) FIXED,
MARGIN_CHOP FIXED,
MP FIXED,
MPI FIXED,

```



```

DMP BIT(8),
TAB BIT(8),
SUB BIT(8),
TRC BIT(8),
ROW BIT(8),
SUP BIT(8),
SAV BIT(8),
UNS BIT(8),
STD BIT(8);

/* ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* ** THE FOLLOWING DECLARATIONS ARE PUNCHED BY THE SEMANTIC PRE-PROCESSOR ** */
/* ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* ** MISCELLANEOUS PROCEDURES ** */
/* ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

PAD: PROCEDURE (STRING,WIDTH) CHARACTER;
      DECLARE STRING CHARACTER, (WIDTH,L) FIXED;
      L = LENGTH (STRING);
      IF L >= WIDTH
        THEN RETURN STRING;
      ELSE RETURN STRING || SUBSTR(X70,0,WIDTH-L);
      END PAD;

I_FORMAT:
PROCEDURE (NUMBER,WIDTH) CHARACTER;
DECLARE (NUMBER,WIDTH,L) FIXED, STRING CHARACTER;
STRING = NUMBER;
L = LENGTH (STRING);
IF L >= WIDTH
THEN RETURN STRING;
ELSE RETURN SUBSTR(X70,0,WIDTH-L) || STRING;
END I_FORMAT;

ERROR:
PROCEDURE (MSG,SEVERITY);
DECLARE MSG CHARACTER, SEVERITY FIXED;
ERROR_COUNT = ERROR_COUNT + 1;
IF ~ CONTROL(BYTE('E'),)

```



```

THEN OUTPUT = I-FORMAT(CARD_COUNT,4) || ' ' || ' ' || BUFFER || ' ' || ' ';
OUTPUT = SUBSTR(POINTER,TEXTLIMIT-CP+MARGIN_CHOP);
OUTPUT = '*** ERROR, MSG || ' ' || LAST_PREVIOUS_ERROR WAS ' || ' ' ||
DETECTED ON LINE || ' ' || PREVIOUS_ERROR || ' ' || '***';
PREVIOUS_ERROR = CARD_COUNT;
IF SEVERITY > 0
THEN IF SEVERE_ERRORS > 25
THEN DO;
OUTPUT = '*** TOO MANY SEVERE ERRORS, CHECKING ABORTED ***';
COMPILING = FALSE;
END;
ELSE SEVERE_ERRORS = SEVERE_ERRORS + 1;
END ERROR;

```

```

HASH:
PROCEDURE(KEY) FIXED;
DECLARE KEY CHARACTER, (I,J,K) FIXED;
K = BYTE(KEY);
DO I = 1 TO LENGTH(KEY) - 1;
J = 8*(I MOD 4);
K = K | SHL(BYTE(KEY,I),J);
END;
RETURN (K MOD HASH_EX);
END HASH;

```

```

GETCELL:
PROCEDURE;
DECLARE I FIXED;
IF FREELIST = 0
THEN CALL ERROR('LIST STRUCTURE OVERFLOW',1);
ELSE DO;
I = FREELIST;
FREELIST = LIST(I);
LIST(I) = 0;
RETURN I;
END;
END GETCELL;

```

```

FREECELL:
PROCEDURE(LOC);
DECLARE LOC FIXED;
LIST(LOC) = FREELIST;
FREELIST = LOC;
END FREECELL;

```

```

ENTER:
PROCEDURE(ENTRY, TABLE);
DECLARE ENTRY CHARACTER, (I,J, TABLE) FIXED;

```


B L A N K


```

DECLARE (I,J,INDEX) FIXED;
ID(INDEX) = 0;
I = HASH(KEY(INDEX));
IF HASH_TABLE(I) = INDEX
THEN DO;
  IF OVERFLOW(INDEX) = NO_OVER
  THEN HASH_TABLE(I) = EMPTY;
  ELSE HASH_TABLE(I) = OVERFLOW(INDEX);
END;
ELSE DO;
  J = HASH_TABLE(I);
  DO WHILE OVERFLOW(J) /= INDEX;
    J = OVERFLOW(J);
  END;
  OVERFLOW(J) = OVERFLOW(INDEX);
END;
OVERFLOW(INDEX) = NO_OVER;
I = POINTR(FREE_KEY);
POINTR(FREE_KEY) = INDEX;
POINTR(INDEX) = I;
END DELETEx;

DELETE:
PROCEDURE(ENTRY, TABLE);
DECLARE ENTRY CHARACTER, (I, TABLE) FIXED;
I = LOOKUP(ENTRY, TABLE);
IF I = 0
THEN CALL ERROR('ENTRY NOT FOUND', 0);
ELSE CALL DELETEx(I);
END DELETE;

POP:
PROCEDURE(INDEX);
DECLARE INDEX FIXED;
STKID(INDEX) = 0;
STKOVr(INDEX) = 0;
STKPTR(INDEX) = FREE_STK;
FREE_STK = INDEX;
END POP;

STACK:
PROCEDURE(ENTRY, STACK);
DECLARE ENTRY CHARACTER, (I, STACK) FIXED;
IF FREE_STK = 0
THEN DO;
  CALL ERROR('MASTER STACK OVERFLOW', 1);
  CALL EXIT;
END;

```


[illegible][illegible]


```

EMITX: PROCEDURE(S);
  DECLARE S FIXED;
  IF CONTROL(BYTE('C'))
  THEN OUTPUT = '***** CODE ' || SYLCNT || ' ' || S || ' *****';
  DO CASE SYLCNT MOD 4;
    /* CASE 0 */
    CODEW = SHL(S,24);
    /* CASE 1 */
    CODEW = CODEW | SHL(S,16);
    /* CASE 2 */
    CODEW = CODEW | SHL(S,8);
    /* CASE 3 */
    DO; CODEW = CODEW | S;
      CODE(SHR(SYLCNT,2)) = CODEW;
      IF CONTROL(BYTE('W'))
      THEN DO;
        S = SHR(SYLCNT,2);
        CALL DUMPCODE(S, CODE(S));
      END;
    END;
  END;
  SYLCNT = SYLCNT + 1;
  IF SYLCNT > MAXSYL
  THEN DO;
    CALL ERROR('CODE TABLE OVERFLOW',2);
    CALL EXIT;
  END;
END; EMITX;

EMIT: PROCEDURE(S);
  DECLARE S FIXED;
  IF S < 128
  THEN CALL EMITX(S); ELSE
  IF S < 256
  THEN DO;

```



```

CALL EMITX(IM1 | "80");
CALL EMITX(S);
END; ELSE
IF S < 65536
THEN DO;
CALL EMITX(IM2 | "80");
CALL EMITX(SHR(S,8));
CALL EMITX(SHL(S,24),24));
END;
ELSE CALL ERROR('LITERAL ' || S || ' IS TOO LARGE',2);
END EMIT;

```

```

/** ** ** ** **
/** ** ** ** *
/** ** ** ** *
/** ** ** ** *
/** ** ** ** *
PROCEDURES PECULIAR TO BALGOL INTERPRETER
/** ** ** ** *
/** ** ** ** *

```

```

GET PRT:
PROCEDURE;
PRTCNT = PRTCNT + 1;
IF PRTCNT > MAXPRT
THEN DO;
CALL ERROR('PRT OVERFLOW',2);
PRTCNT = 0;
END;
CONTROL(BYTE('P'))
THEN OUTPUT = '***** PRT CELL ' || PRTCNT - 1 || ' HAS BEEN ASSIGNED *****';
DATA(PRTCNT-1) = 0;
RETURN PRTCNT-1;
END GETPRT;

```

```

GET STORAGE:
PROCEDURE(N);
DECLARE (I,N) FIXED;
I = DATACNT;
DATACNT = DATACNT + N;
IF DATACNT > MAXDATA - PRTCNT
THEN DO;
CALL ERROR('DATA TABLE OVERFLOW',2);
I = 0;
DATACNT = 0;
END;
DO N = MAXDATA - DATACNT TO MAXDATA - I - 4;
DATA(N) = 0;
END;

```



```

RETURN I;
END GETSTORAGE;

STORESTRING:
PROCEDURE(S);
DECLARE S CHARACTER, (I,J,K,L,M,T) FIXED;
J = LENGTH(S);
I = GETSTORAGE(SHR(J+4,2));
T = SHL(J,24);
L = I;
M = MAXDATA - I;
DO K = 0 TO J - 1;
  T = T | SHL(BYTE(S,K),SHL(3-L,3));
  L = (L + 1) & "3";
  IF L = 0
  THEN DO;
    DATA(M) = T;
    T = 0;
    M = M - 1;
  END;
END;
IF L /= 0
THEN DATA(M) = T;
END STORESTRING;

STUFFBACK:
PROCEDURE(REF,LOC);
DECLARE (REF,LOC,I,T) FIXED;
I = SHR(REF,2);
T = CODE(I);
DO CASE (REF & "3");
/* CASE 0 */
DO; REF = SHR(T,16);
  CODE(I) = (T & "FFFF") | SHL(LOC,16);
END;
/* CASE 1 */
DO; REF = SHR(T,8) & "FFFF";
  CODE(I) = (T & "FF0000FF") | SHL(LOC,8);
END;
/* CASE 2 */

```



```

DO; REF = T & "FFFF";
CODE(I) = (T & "FFFFFF0000") | LOC;
END;

/* CASE 3 */

DO; REF = SHL((T & "FF"),8) | SHR(CODE(I+1),24);
CODE(I) = (T & "FFFFFFF000") | SHR(LOC,8);
T = CODE(I+1);
CODE(I+1) = SHL((LOC & "FF"),24) | (T & "FFFFFFF");
END;

END;
IF CONTROL(BYTE('S'))
THEN DO;
OUTPUT = '***** BACKSTUFFING *****';
CALL DUMPCODE(I,CODE(I));
CALL DUMPCODE(I+1,CODE(I+1));
END;
RETURN REF;
END STUFFBACK;

```

```

DUMPFIL:
PROCEDURE;
I1 = DATA#;
DO WHILE I1 > 3;
DATA(I1) = DATA(I1-4);
I1 = I1 - 1;
END;
SYLCNT = (SYLCNT + 3)/4;
DATA(0) = PRTCNT;
DATA(1) = DATACNT;
DATA(2) = SYLCNT;
DATA(3) = 200;
I1 = PRTCNT + 4;
I2 = DATA#;
DO WHILE I1 < I2;
I3 = DATA(I1);
DATA(I1) = DATA(I2);
DATA(I2) = I3;
I1 = I1 + 1;
I2 = I2 - 1;
END;
I1 = DATACNT + PRTCNT + 4;
I4 = 0;
I2 = SYLCNT - 1;

```



```

CP = 0;
END;
DO CASE CHARTYPE(BYTE(TEXT));
/* CASE 0 -- ILLEGAL CHARACTER */
DO; CALL ERROR('ILLEGAL CHARACTER: ' || SUBSTR(TEXT,0,1),0);
END;
/* CASE 1 -- RESERVED WORD OR IDENTIFIER */
DO FOREVER;
DO CP = CP + 1 TO TEXT_LIMIT;
IF NOT LETTER_OR_DIGIT(BYTE(TEXT,CP))
THEN DO;
IF CP > 0
THEN BCD = BCD || SUBSTR(TEXT,0,CP);
S1 = LENGTH(BCD);
IF S1 > 1
THEN IF S1 <= RESERVED_LIMIT
THEN DO I = V_INDEX(S1-1) TO V_INDEX(S1) - 1;
IF BCD = V(I)
THEN DO;
TOKEN = I;
RETURN;
END;
END;
TOKEN = IDENT;
RETURN;
END;
END;
BCD = BCD || TEXT;
CALL GET_CARD;
CP = -1;
END;
/* CASE 2 -- NUMBER */
DO; TOKEN = NUMBER;
DO FOREVER;
DO CP = CP TO TEXT_LIMIT;
S1 = BYTE(TEXT,CP);
IF S1 < "F0"
THEN RETURN;
NUMBER_VALUE = 10 * NUMBER_VALUE + S1 - "F0";
END;

```



```

        CALL GET_CARD;
    END;
END;
/* CASE 3 -- SPECIAL CHARACTER */
DO;
    TOKEN = TX(BYTE(TEXT));
    CP = 1;
    RETURN;
END;
/* CASE 4 -- BLANK */
DO;
    CP = 1;
    DO WHILE BYTE(TEXT,CP) = BYTE(' ') & CP <= TEXT_LIMIT;
        CP = CP + 1;
    END;
    CP = CP - 1;
END;
/* CASE 5 -- MULTIPLICATION OR EXPONENTIATION */
DO;
    CP = 1;
    IF BYTE(TEXT,CP) ^= BYTE('*')
    THEN TOKEN = MULT;
    ELSE DO;
        TOKEN = EXPON;
        CP = CP + 1;
    END;
    RETURN;
END;
/* CASE 6 -- COMMENT OR DIVIDE */
DO;
    CALL CHAR;
    IF BYTE(TEXT,CP) ^= BYTE('*')
    THEN DO;
        TOKEN = DIVIDE;
        RETURN;
    END;
    S1, S2 = BYTE(' ');
    DO WHILE S1 ^= BYTE('*') | S2 ^= BYTE('/');
        IF S1 = BYTE('$')
        THEN DO;

```



```

CONTROL(S2) = ~ CONTROL(S2);
IF S2 = BYTE('T')
THEN CALL TRACE; ELSE
IF S2 = BYTE('U')
THEN CALL UNTRACE; ELSE
IF S2 = BYTE('I')
THEN IF CONTROL(S2)
THEN MARGIN_CHOP = TEXT_LIMIT - CP + 1;
ELSE MARGIN_CHOP = 0;
END; S2 = S2;
CALL CHAR;
S2 = BYTE(TEXT,CP);
END;
END;

/* CASE 7 -- CHARACTER STRING */
DO FOREVER; STRING;
TOKEN = STRING;
S1 = 1; CP = CP + 1;
DO WHILE BYTE(TEXT,CP) ~= BYTE('');
IF CP <= TEXT_LIMIT
THEN CP = CP + 1;
ELSE DO;
IF LENGTH(BCD) + CP > 257
THEN DO;
CALL ERROR('CHARACTER STRING TOO LONG',0);
RETURN;
END;
IF CP > S1
THEN BCD = BCD || SUBSTR(TEXT,S1,CP-S1);
TEXT = X1;
CP = 0;
CALL GET_CARD;
S1 = 0;
END;
END;
IF LENGTH(BCD) + CP > 257
THEN DO;
CALL ERROR('CHARACTER STRING TOO LONG',0);
RETURN;
END;
IF CP > S1
THEN BCD = BCD || SUBSTR(TEXT,S1,CP-S1);
CALL CHAR;
IF BYTE(TEXT,CP) ~= BYTE('')

```



```

THEN STOPIT(I) = TRUE; ELSE
IF S = '*' THEN
MULT = I; ELSE
IF S = '**' THEN
EXPON = I; ELSE
IF S = ':' THEN
COLON = I; ELSE
IF S = '::=' THEN
EQLS = I;
END;
IF IDENT = NT
THEN RESERVED_LIMIT = LENGTH(V(NT-1));
ELSE RESERVED_LIMIT = LENGTH(V(NT));
V(EOFIL) = 'EOF'; TRUE;
DO I = 0 TO 255;
IF NOT_LETTER_OR_DIGIT(I) = TRUE;
END;
DO I = 0 TO LENGTH(ALPHABET) - 1;
J = BYTE(ALPHABET,I);
TX(J) = I; OR_DIGIT(J) = FALSE;
CHARTYPE(J) = 1;
END;
DO I = 0 TO 9;
J = BYTE('0123456789',I);
NOT_LETTER_OR_DIGIT(J) = FALSE;
CHARTYPE(J) = 2;
END;
DO I = V_INDEX(0) TO V_INDEX(1) - 1;
J = BYTE(V(I));
TX(J) = I;
CHARTYPE(J) = 3;
END;
CHARTYPE(BYTE(' ')) = 4;
CHARTYPE(BYTE('*')) = 5;
CHARTYPE(BYTE('/')) = 6;
CHARTYPE(BYTE(':')) = 7;
CHARTYPE(BYTE(':')) = 8;
CP = 0;
TEXT_LIMIT = -1;
TEXT = '';
CONTROL(BYTE('L')) = TRUE;
CALL SCAN;
SP = 1;
STACK(SP) = EOFIL;
PARSE = 0 TO HASH_EX;
DO I = 0 TO HASH_TABLE(I) - 1;

```



```

END;
DO I = 0 TO MAXKEY;
  ID(I) = 0;
  OVERFLOW(I) = NO_OVER;
  POINTNR(I) = I + 1;
END;
POINTNR(MAXKEY) = 0;
DO I = 0 TO MAXSTK;
  STKID(I) = 0;
  STKQVR(I) = NO_OVER;
  STKPTR(I) = I + 1;
END;
STKPTR(MAXSTK) = 0;
DO I = 0 TO FC_MAX;
  CODELOC(I) = 0;
  FLOPOVER(I) = I + 1;
END;
FLOPOVER(FC_MAX) = 0;
DO I = 0 TO MAXLIST;
  LIST(I) = I + 1;
END;
LIST(MAXLIST) = 0;
DO I = 0 TO MAXLABEL;
  LABELADD(I) = 0;
  LABELOVER(I) = 0;
END;

```

```

/** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** 
/** ** ** ** ** 
/** ** ** ** 
/** ** ** ** 

```

BALGOL INITIALIZATION

```

I2 = ADDR(ADD);
I3 = 1;
DO WHILE I3 <= OPERATORS;
  COREBYTE(I2) = I3;
  I2 = I2 + 1;
  I3 = I3 + 1;
END;
END INITIALIZATION;

```

```

/** ** ** ** 
/** ** ** ** 
/** ** ** ** 
/** ** ** ** 

```

MISCELLANEOUS PROCEDURES


```

FINDLABEL:
PROCEDURE (ENTRY); CHARACTER, I FIXED;
DECLARE ENTRY LABELPTR;
DO I = 0 TO LABELPTR;
IF LABELS(I) = ENTRY
THEN RETURN I;
END;
RETURN 0;
END FINDLABEL;

SAVELABEL:
PROCEDURE (ENTRY);
DECLARE ENTRY CHARACTER, (I,J,LOC) FIXED;
I = FINDLABEL(ENTRY);
IF I = 0
THEN DO;
IF LABELPTR > MAXLABEL
THEN CALL ERROR('TOO MANY PROGRAM LABELS',1);
ELSE DO;
LABELS(LABELPTR) = ENTRY;
LABELADD(LABELPTR) = SYLCNT;
LABELPTR = LABELPTR + 1;
END;
END;
ELSE DO;
IF LABELADD(I) = 0
THEN DO;
J = LABELOVER(I);
DO WHILE J /= 0;
LOC = LIST(J) & "FFFFFFF";
CALL STUFFBACK(LOC,SYLCNT);
CALL FREECELL(J);
J = SHR(LIST(J),24);
END;
LABELADD(I) = SYLCNT;
LABELOVER(I) = 0;
END;
ELSE LABELADD(I) = SYLCNT;
END;
END;
END SAVELABEL;

RESTORELABEL:
PROCEDURE (ENTRY);
DECLARE ENTRY CHARACTER, (I,J) FIXED;
I = FINDLABEL(ENTRY);

```



```

CALL EMIT(0);
CALL EMIT(0);
END;
END SAVELOC;

RESTORELOC;
PROCEDURE(ENTRY);
DECLARE ENTRY CHARACTER, (I,SAVE) FIXED;
I = FC_PTR;
SAVE = I;
DO WHILE(FLOP(I) /= ENTRY & I /= 0);
  SAVE = I;
  I = FLOPOVER(I);
END;
IF I = 0
THEN CALL ERROR('INVALID STUFFBACK PARAMETER',1);
ELSE DO;
  CALL STUFFBACK(CODELOC(I),SYLCNT);
  CODELOC(I) = 0;
  FLOPOVER(SAVE) = FLOPOVER(I);
  IF SAVE = I
  THEN FC_PTR = FLOPOVER(I);
  FLOPOVER(I) = FC_FREE;
  FC_FREE = I;
END;
END RESTORELOC;

DUMPIT:
PROCEDURE;
DOUBLE SPACE;
OUTPUT = 'STACKING DECISIONS = ' || CALLCOUNT(1);
OUTPUT = 'SCAN = ' || CALLCOUNT(3);
OUTPUT = 'EMIT = ' || CALLCOUNT(4);
OUTPUT = 'FREE STRING AREA = ' || FREELIMIT - FREEBASE;
END DUMPIT;

STACK_DUMP:
PROCEDURE;
DECLARE LINE CHARACTER;
LINE = 'PARTIAL PARSE TO THIS POINT IS: ';
DO I = 2 TO SP;
  IF LENGTH(LINE) > 105
  THEN DO;
    OUTPUT = LINE;
    LINE = X4;
  END;
  LINE = LINE || X1 || V(PARSE_STACK(I));
END;

```



```

OUTPUT = LINE;
END STACK_DUMP;

```

```

/** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** 
/** ** ** ** ** 
/** ** ** ** 
/** ** ** ** 

```

THE SYNTHESIS ALGORITHM

```

SYNTHESIZE:
PROCEDURE (PRODUCTION_NUMBER);
DECLARE PRODUCTION_NUMBER FIXED;
DO CASE PRODUCTION_NUMBER;

```

```

/* CASE 0 */ DO; IF MP ^= 2 THEN DO; CALL ERROR('EOF AT INVALID POINT',1);
/* CASE 1 */ DO; IF MP ^= 2 THEN DO; CALL ERROR('EOF AT INVALID POINT',1);
CALL STACK_DUMP; END; COMPILING=FALSE; DO I1 = 0 TO LABELPTR; IF LABELOVER(I1)
^= 0 THEN CALL ERROR('UNSATISFIED LABEL: ', I1 LABELS(I1),1); END; DO I1 = 0
TO FC MAX; IF CODELOC(I1) ^= 0 THEN CALL ERROR('UNSATISFIED STUFFBACK', I1
, PARAMETER: ' || FLOP(I1),1); END;

```

```

/** ** ** ** 
/** ** ** ** 
/** ** ** ** 
/** ** ** ** 

```

THE FOLLOWING CODE STREAM IS PUNCHED BY THE SEMANTIC PRE-PROCESSOR

```

END;
END SYNTHESIZE;

```

```

/** ** ** ** 
/** ** ** ** 
/** ** ** ** 
/** ** ** ** 

```

SYNTACTIC PARSING FUNCTIONS

```

RIGHT_CONFLICT:
PROCEDURE (LEFT) BIT(1);
DECLARE LEFT FIXED;
RETURN ("CO" & SHL(BYTE(C1(LEFT), SHR(TOKEN,2)), SHL(TOKEN,1) & "06"))
= 0;
END RIGHT_CONFLICT;

```



```

RECOVER:
PROCEDURE:
IF - FAILSOFT
THEN CALL SCAN;
FAILSOFT = FALSE;
DO WHILE - STOPIT(TOKEN);
CALL SCAN;
END;
DO WHILE RIGHT_CONFLICT(PARSE_STACK(SP));
IF SP > 2
THEN SP = SP - 1;
ELSE CALL SCAN;
END;
OUTPUT = 'RESUME:' || SUBSTR(POINTER,TEXT_LIMIT-CP+MARGIN_CHOP+7);
END RECOVER;

STACKING:
PROCEDURE BIT(1);
CALLCOUNT(1) = CALLCOUNT(1) + 1;
DO FOREVER;
DO CASE
SHR(BYTE(C1(PARSE_STACK(SP)),SHR(TOKEN,2)),SHL(3-TOKEN,1)
& 6) & 3;
/* CASE 0 - ILLEGAL SYMBOL PAIR */
DO;
CALL ERROR('ILLEGAL SYMBOL PAIR; ' || V(PARSE_STACK(SP)) ||
X1 || V(TOKEN),1);
CALL STACK_DUMP;
CALL RECOVER;
END;
/* CASE 1 - STACK TOKEN */
RETURN TRUE;
/* CASE 2 - DON'T STACK TOKEN */
RETURN FALSE;
/* CASE 3 - CHECK TRIPLES */
DO;
J = SHL(PARSE_STACK(SP-1),16) + SHL(PARSE_STACK(SP),8) +
TOKEN;
I = -1;
K = NCITRIPLES + 1;
DO WHILE I + 1 < K;

```



```

L = SHR(I+K,1);
IF C1TRIPLES(L) > J
THEN K = L; ELSE
IF C1TRIPLES(L) < J
THEN I = L;
ELSE RETURN TRUE;
END;
RETURN FALSE;
END;

```

```

END;
END STACKING;

```

```

PR_OK: PROCEDURE (PRD) BIT(1);
DECLARE (H,I,J,PRD) FIXED;
DO CASE CONTEXT_CASE (PRD);
/* CASE 0 - NO CHECK REQUIRED */

```

```

RETURN TRUE;

```

```

/* CASE 1 - RIGHT CONTEXT CHECK */

```

```

RETURN ~ RIGHT_CONFLICT(HDTB (PRD));

```

```

/* CASE 2 - LEFT CONTEXT CHECK */

```

```

DO; H = HDTB (PRD) - NT;
I = PARSE_STACK (SP-PRLENGTH (PRD));
DO J = LEFT_INDEX (H-1) TO LEFT_INDEX (H) - 1;
IF LEFT_CONTEXT (J) = I
THEN RETURN TRUE;
END;

```

```

RETURN FALSE;

```

```

END;
/* CASE 3 - CHECK TRIPLES */

```

```

DO; H = HDTB (PRD) - NT;
I = SHL (PARSE_STACK (SP-PRLENGTH (PRD)), 8) + TOKEN;
DO J = TRIPLE_INDEX (H-1) TO TRIPLE_INDEX (H) - 1;
IF CONTEXT_TRIPLE (J) = I
THEN RETURN TRUE;
END;

```


[illegible]

```

MP = SP - PLENGTH(PRD) + 1;
MPPI = MP + 1;
CALL SYNTHESIZE (PRDTB(PRD));
SP = MP;
PARSESTACK(SP) = HDTB(PRD);
RETURN;

```

```
END;  
CALL STACK_DUMP;  
CALL SOFT = FALSE;  
CALL RECOVER;  
END REDUCE;
```

CHECKING ABORTED ***',2);

THE BALGOL-2 INTERPRETER

[illegible]

```

DECLARE
RECORD (RECSIZE) FIXED,
M(SIZE) FIXED,
PARM(3) FIXED,
(I,J,K,L,R,N,
U,DUMPV,
/* HOLD ONE LINE FROM FILE1 (CODE FILE) */
/* MEMORY OF SIMULATED BALGOL-2 MACHINE */
/* LOADING PARAMETERS (DEFINED) */
/* TEMPORARY GLOBAL VARIABLES */

```



```

F,
RS,
RG,
RC,
RP,
RT,
SB,
B,
D,
T,
TRACE,
FIELDW,
S4,
(C1,
F1,
MESS,
MEMLOC,
UNDERSCORE,
PTOG,
DECLARE,
OPCODE,
CHARSET,
BLANK,
(OBUFF,
/* BASE OF FREE STORAGE */
/* REGISTER S (STACK REGISTER) */
/* REGISTER G (GOSUB REGISTER) */
/* REGISTER C (CONTROL REGISTER) */
/* REGISTER P (PROGRAM REGISTER--ONE CODE WORD) */
/* REGISTER T (TEMPORARY REGISTER--ONE SYLLABLE) */
/* STACK LIMITS REGISTERS (STACK TOP, BOTTOM) */
/* BASE OF CODE AND BASE OF DATA */

/* IF NON ZERO THEN MACHINE PRINTS TRACE */

F1, F2, F3, F13, F23) FIXED,
(C1, C2) CHARACTER,
F1, F2, F3, F12, F13, F23) FIXED,
(MESS(MAXMESS)) CHARACTER,
MEMLOC CHARACTER,
UNDERSCORE CHARACTER,
PTOG BIT(1);
/* OBP: OUTPUT BUFFER POINTER */
DECLARE OBP FIXED;
OPCODE CHARACTER,
CHARSET CHARACTER,
BLANK CHARACTER INITIAL(' ');
/* INPUT AND OUTPUT BUFFERS */
(OBUFF, IBUFF) CHARACTER; /* FIELD SIZE BLANKS */

/* STORAGE ALLOCATION PROCEDURES */
INITMEM: PROCEDURE;
/* INITIALIZE MASKS */
F1 = "40000000"; /* FIELD 1: BIT 30 */
F2 = "3FFFF8000"; /* FIELD 2: BITS 15 - 29 */
F3 = "7FFFF"; /* FIELD 3: BITS 0 - 14 */
F12 = F1 | F2; F13 = F1 | F3; F23 = F2 | F3;
/* INITIALIZE MEMORY */
/* NOTE THAT GLOBAL VARIABLE I IS BEING USED */
DO I=F TO SIZE; M(I)=0;
END;
M(F)=F+2; M(F+1)= F1|SIZE; M(SIZE)=SHL(F+1,15);
M(F+2)=SHL(F,15);
OUTPUT=MEMEM;
END INITMEM;
AVAIL: PROCEDURE(S);
DECLARE (S,A) FIXED;
A=M(F);
IF A=0 THEN
  M(A)=SHL(S,15)|(M(A)&F3);
  M(S)=SHL(F,15)|A; M(F)=S;
END AVAIL;
LINKIN: PROCEDURE(LINKA, LINKB, S);
DECLARE (I,J, LINKA, LINKB, S) FIXED;

```



```

I=LINKB-LINKA-S;
IF I>2 THEN
DO; I=LINKA+S+1;
M(LINKA)=(F2&M(LINKA))|I; /* FORWARD LINK */
M(I)=F1|SHL(LINKA,15)|LINKB;
LINKA=I; J=M(LINKB);
M(LINKB)=SHL(LINKA,15)|(F3&J);
CALL AVAIL(LINKA+1);
END; ELSE
M(LINKA)=M(LINKA)&F23; /* AVAIL BIT TURNED OFF */
END LINKIN;
GETSPACE: PROCEDURE (S) FIXED;
DECLARE (S,I,J,LINKA,LINKB,AVA) FIXED;
AVA=M(F);
DO WHILE AVA=0;
LINKA=AVA-1; LINKB=M(LINKA)&F3;
IF LINKB-LINKA>S THEN
DO; J=M(AVA); I=SHR(J,15); J=J&F3;
M(I)=(M(I)&F2)|J;
IF J=0 THEN
M(J)=SHL(I,15)|(M(J)&F3);
CALL LINKIN(LINKA,LINKB,S);
RETURN LINKA+1;
END;
AVA=M(AVA)&F3;
END;
RETURN -1;
END GETSPACE(S);
RELEASE: PROCEDURE (LINK,S,I,J) FIXED;
LINK=M(S); I=SHR(LINK,15); J=LINK&F3;
M(I)=(M(I)&F12)|J;
IF J=0 THEN
M(J)=SHL(I,15)|(M(J)&F3);
END RELEASE;
FORGETSPACE: PROCEDURE (L);
DECLARE (L,I,LINKI,J,LINKJ,K,LINKK) FIXED;
LINKK=0; I=L-1; LINKI=M(I); J=LINKI&F3;
LINKJ=M(J); K=SHR(LINKI,15);
IF K=0 THEN LINKK=M(K);
IF SHR(LINKK|LINKJ,30)=0 THEN
DO;
CALL AVAIL(L); M(I)=F1|LINKI;
END; ELSE
DO;
SHR(LINKJ,30)>0 THEN
DO; CALL RELEASE(J+1); J=LINKJ&F3; LINKJ=M(J);
END;

```



```

IF SHR(LINKK,30)>0 THEN
DO: I=K; LINKI=LINKK;
END; ELSE
DO: LINKI=LINKI|F1; CALL AVAIL(I+1);
END;
M(I)=(F12&LINKI)|J;
M(J)=SHL(I,15)|(F3&LINKJ);
END;
END FORGETSPACE;
/* END OF STORAGE ALLOCATION PROCEDURES */
INIT: PROCEDURE;
/* K COUNTS LOCATIONS IN M, L COUNTS LINES IN FILE 1 AS IT IS BEING
UNLOADED, AND R COUNTS CELLS IN RECORD */
K,L=0; R=NPARM+1; /* GET INITIAL PARAMETERS FOR LOADING MEMORY */
RECORD=FILE(CODE,0); PARM(I)=RECORD(I);
DO I=0 TO NPARM;
END;
D=#PRT; /* D: BASE OF THE DATA (FIXED) */
B=D+#DATA; /* B: BASE OF THE CODE */
SB=B+#CODE; ST=SB+#STACK;
OUTPUT=#PRT; DATA=#DATA; CODE=#CODE;
MEMLOC=#MEMORY ADDR; FIXFORM=#FIXED FORMAT;
UNDERSCORE=#;
/* LOAD PRT, DATA, AND CODE */
DO I=0 TO NPARM-1; N=PARM(I)-1;
DO J=0 TO N;
IF R > RECSIZE THEN
/* READ ANOTHER RECORD FROM FILE 1 */
DO: R=0; L=L+1; RECORD=FILE(CODE,L);
END;
M(K)=RECORD(R); R=R+1; K=K+1;
END;
END;
/* INITIALIZE FREE STORAGE */
F=ST; /* BEGINNING OF FREE STORAGE */
CALL SET REGISTERS /*
B4=B*4; S4=SB*4; TRACE,DUMPV=0;
RC=B4; RS=SB+1; RA,RB=0; RP,RT=0; RG=ST; INVALID;
IBUFF,OBUFF=; OBP=0;
FIELDWIDTH=15;
CHARSET=
/* 000 -031 */
/* 032 -063 */
/* 064 -095 */
/* 096 -127 */
/* 128 -159 */
/* INITIALIZE CHARACTER SET TO EBCDIC CHARACTERS */
'<(+|&
'&_>? $*):~'
' -/ ABCDEFGHI JKLMNOPQR :#@;=".,:|'

```



```

160 -191 **
192 -223 **
224 -255 **
MESS(0)='INVALID OPERATION CODE (OP CODE C)';
MESS(1)='STACK UNDERFLOW';
MESS(2)='STACK UNDERFLOW';
MESS(3)='STACK OVERFLOW';
MESS(4)='INVALID RC SETTING (< FIXED DATA)';
MESS(5)='INVALID RC SETTING (> STACK BASE)';
MESS(6)='INVALID OPERATION CODE (TOO LARGE)';
MESS(7)='LOWER BOUND EXCEEDS UPPER BOUND';
MESS(8)='END OF FILE ENCOUNTERED ON INPUT';
MESS(9)='INVALID CHARACTER IN INPUT STREAM';
MESS(10)='ATTEMPTED DIVISION BY ZERO';
MESS(11)='EXONENTIATION ERROR: X**Y, X<';
MESS(12)='REGISTER S=RETURN WITH NO PREVIOUS CALL';
MESS(13)='ATTEMPT TO RETURN WITH NO PREVIOUS CALL';
MESS(14)='FREE STORAGE EXHAUSTED';
MESS(15)='FREE STORAGE EXHAUSTED';
MESS(16)='INVALID SUBSCRIPT';
MESS(17)='INVALID INTRINSIC FUNCTION';
MESS(18)='INVALID TAB EXPRESSION';
MESS(19)='INVALID FIELD WIDTH PARAMETER';
OPCODE =
/* LITERAL CALL SYLLABLE */ 'LIT' ||
/* 08 ARITHMETIC OPERATORS */ 'ADDIMMULDIVEXPNEGRNDTRU' ||
/* 09 BOOLEAN OPERATORS */ 'LSSELEQLNEQEQGTRNOTANDBOR' ||
/* 04 LOAD/STORE OPERATORS */ 'IMIM2LODSTO' ||
/* 07 STACK CONTROL */ 'PRORTNXITDELDOUPXCHINX' ||
/* 14 MONITOR CALLS */ 'BFNBBNBFCB8CBRSBSCNOP' ||
/* MOVE THE STORE DESTRUCTIVE (LAST OP) TO THE PROPER PLACE LATER */
/* PAGE EJECT */
OUTPUT(1)='1';
RETURN;
END INIT;

PRINT: PROCEDURE;
/* DUMP THE OUTPUT BUFFER */
DO WHILE LENGTH(OBUFF) < PRINTSIZE; OBUFF=OBUFF||BLANK;
END;
OUTPUT=SUBSTR(OBUFF,0,PRINTSIZE); CBP=0; OBUFF='';
END PRINT;
MEMDUMP: PROCEDURE;
DECLARE (I,J,K) FIXED, Z4 CHARACTER INITIAL('0000');
DECLARE X CHARACTER , (R,N,L, LAST,AW,LW) FIXED;
OUTPUT=BLANK;

```



```

DO: IF L<128 THEN N=1; ELSE C2=SUBSTR(OPCODE, (L&"7F")*3,3);
END;
IF N>0 THEN
DO: C2=L; N=N-1;
IF LENGTH(C2)<3 THEN C2=SUBSTR(Z4,0,3-LENGTH(C2))||C2;
END; ELSE
IF C2='IM1' THEN N=1; ELSE IF C2='IM2' THEN N=2;
C1=C1||C2||';';
END;
OUTPUT=X||C1;
END;
LAST=J;
END;
X=BLANK;
OUTPUT=X||MEMLOC||FIXFORM; X=X||';'; K=ST-1;
DO I=SB TO K; J=M(I);
IF LAST=J THEN
DO: C1=I; C1=C1||SUBSTR(BLANK,0,15-LENGTH(C1));
OUTPUT=X||C1||J;
END;
LAST=J;
END;
X=BLANK;
OUTPUT=X||MEMLOC||FIXFORM||'FREE STORAGE AREA (FSA)'; X=X||BLANK;
AW=F; LW=F+1;
DO I=F TO SIZE; J=M(I);
IF (LAST=J) | (I=LW) | (I=AW) THEN
DO: C1=I; C1=C1||SUBSTR(BLANK,0,15-LENGTH(C1));
C2=J; C2=C2||SUBSTR(BLANK,0,15-LENGTH(C2)); C1=C1||C2;
IF SHR(J,30) THEN C1=C1||'A'; ELSE C1=C1||'I';
C2=SHR(J,15)&"7FFF";
IF LENGTH(C2)<5 THEN C2=SUBSTR(Z4,0,5-LENGTH(C2))||C2;
C1=C1||C2||';';
C2=J&"7FFF";
IF LENGTH(C2)<5 THEN C2=SUBSTR(Z4,0,5-LENGTH(C2))||C2;
C1=C1||C2||';';
IF LW=I THEN
DO: C1=C1||'LINK WORD'; LW=J&"7FFF";
END; ELSE
IF AW=I THEN
DO: C1=C1||'AVAIL WORD'; AW=J&"7FFF";
END;
OUTPUT=X||C1;
END;
LAST=J;
END;
OUTPUT=BLANK;
END MEMDUMP;
OUTPUT=BLANK;
END OF DUMP.; OUTPUT=BLANK;

```



```

ERROR: PROCEDURE(E);
DECLARE E FIXED;
IF LENGTH(OBUFF)>0 THEN CALL PRINT;
OUTPUT='** EXECUTION ERROR: '||MESS(E)||', AT SYLLABLE'||
(RC-B4)||, **';
IF DUMPV=1 THEN CALL MEMDUMP;
CALL EXIT;
END ERROR;

STRING: PROCEDURE CHARACTER;
I=RA; J=M(I); K=0; L=SHR(J,24); C2='';
/* NOTE THAT THE GLOBAL VARIABLES I,J,K,L, AND C2 ARE BEING USED */
DO WHILE L>0; K=K+8;
IF K=32 THEN
DO; I=I+1; J=M(I); K=0;
/* I CONTAINS NEXT GROUP OF CHARACTERS */
END;
N=SHL(J,K); N=SHR(N,24);
C2=C2||SUBSTR(CHARSET,N,1); L=L-1;
END;
RETURN C2;
END STRING;

WRITE: PROCEDURE(T);
DECLARE T BIT(1); /* T=0 => NUMERIC WRITE FROM RA, T=1 => CHARACTER WRITE */
IF T=0 THEN /* NUMERIC */
C1=RA; ELSE /* CHARACTER */
C1=STRING;
L=LENGTH(OBUFF); J=LENGTH(C1);
IF L>OBP+J THEN /* GET REMAINING SEGMENT OF BUFFER */
C2=SUBSTR(OBUFF,OBP+J); ELSE C2='';
IF OBP>0 THEN OBUFF=SUBSTR(OBUFF,O,OBP); ELSE OBUFF='';
OBUFF=OBUFF||C1||C2;
OBP=OBP+J;
L=LENGTH(OBUFF); J=FIELDWIDTH-(OBP MOD FIELDWIDTH);
IF J=FIELDWIDTH THEN OBP=OBP+J;
IF OBP>L THEN OBUFF=OBUFF||SUBSTR(BLANK,0,OBP-L);
DO WHILE LENGTH(OBUFF)>PRINTSIZE;
OUTPUT=SUBSTR(OBUFF,0,PRINTSIZE); OBUFF=SUBSTR(OBUFF,PRINTSIZE);
END;
IF OBP>LENGTH(OBUFF) THEN OBP=LENGTH(OBUFF);
END WRITE;

TABULATE: PROCEDURE;
I=RA-1; /* TAB TO COLUMN RA */
IF (I<0) | (I>PRINTSIZE) THEN CALL ERROR(18);
DO WHILE I>LENGTH(OBUFF);
J=I-LENGTH(OBUFF);

```



```

IF J>FIELDWIDTH THEN J=FIELDWIDTH;
OBUFF=OBUFF||SUBSTR(BLANK,C,J);
END;
OBP=I;
IF OBP > PRINTSIZE THEN CALL PRINT;
END TABULATE;

READLINE: PROCEDURE;
IBUFF=INPUT;
IF LENGTH(IBUFF)=0 THEN CALL ERROR(8);
END READLINE;

READER: PROCEDURE FIXED;
IF LENGTH(IBUFF)=0 THEN CALL READLINE;
DO WHILE SUBSTR(IBUFF,0,1)=' '; IBUFF=SUBSTR(IBUFF,1);
IF LENGTH(IBUFF)=0 THEN
CALL READLINE;
END;
I=1; J=0; L=LENGTH(IBUFF);
IF SUBSTR(IBUFF,0,1)='-.' THEN
DO; I=-1; IBUFF=SUBSTR(IBUFF,1);
END;
DO WHILE SUBSTR(IBUFF,0,1)~=' ';
L=BYTE(SUBSTR(IBUFF,0,1))-240;
IF (L<0)|(L>9) THEN CALL ERROR(9);
J=J*10+L;
IBUFF=SUBSTR(IBUFF,1); IF LENGTH(IBUFF)=0 THEN CALL READLINE;
END;
RETURN I*J;
END READER;

TRACEHEAD: PROCEDURE;
TRCNT=0;
OUTPUT=BLANK||
' IOP CODE I REGISTER A I REGISTER B I REG S I REG C I SYL I';
END TRACEHEAD;
TRACER: PROCEDURE (OP);
DECLARE OP CHARACTER, I FIXED;
IF TRCNT > TRCMAX THEN CALL TRACEHEAD;
I=7-LENGTH(OP);
IF I>0 THEN OP=OP||SUBSTR(UNDERSCORE,0,I); C1=BLANK||'|'||OP;
C2=RA; I=12-LENGTH(C2);
IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I); C1=C1||'|'||C2;
C2=RB; I=12-LENGTH(C2);
IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I); C1=C1||'|'||C2;
C2=RS; I=5-LENGTH(C2);
IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I); C1=C1||'|'||C2;
C2=RG; I=5-LENGTH(C2);

```



```

IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I); C1=C1||'|'||C2;
C2=RC; I=6-LENGTH(C2);
IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I); C1=C1||'|'||C2;
C2=RC-B4; I=6-LENGTH(C2);
IF I>0 THEN C2=C2||SUBSTR(UNDERSCORE,0,I);
OUTPUT=C1||'|'||C2||'|'; TRCNT=TRCNT+1;
END TRACER;
GETCNE: PROCEDURE (C);
DECLARE C FIXED;
I=C MOD 4; J=SHR(C,2); J=M(J);
RETURN SHR ( SHL( J,I*8) ,24 );
END GETCNE;
GETTWO: PROCEDURE (C);
DECLARE (C,J) FIXED;
J=C MOD 4;
IF J=3 THEN RETURN SHL(GETONE(C),8) | GETONE(C+1);
C=M(SHR(C,2)); J=SHL(J,3); RETURN SHR(SHL(C,J),16);
END GETTWO;

EXECUTE: PROCEDURE;
/* MAIN PROCEDURE OF BALGOL-2 MACHINE -- SIMULATES MACHINE'S CPU */
CALL INIT; /* LOADS PROGRAM FROM FILE 1 */
DO WHILE RC=0; /* RC SET TO ZERO BY XIT INSTRUCTION */
IF (RC & "3") =0 THEN INVALID;
IF INVALID THEN
DO; RP=M(SHR(RC,2)); /* RP HOLDS 4 SYLLABLES */
I=SHL(RC & "3",3);
IF I=C THEN RP=SHR(SHL(RP,I),I);
VALID;
END;
I=RC & "3";
RT=SHR(RP, SHL(3-I,3)); I=SHL(I+1,3);
RP=SHR(SHL(RP,I),I);
/* CHECK FOR POSSIBLE ERROR CONDITIONS */
IF RSK=SB THEN CALL ERROR(2);
IF RS>=ST THEN CALL ERROR(3);
IF RC<B4 THEN CALL ERROR(4);
/* CHECK FOR OPERATOR TYPE */
IF RT<128 THEN /* LITERAL CALL SYLLABLE */
DO; PUSH; I=RT; RA=I; STEP;
TEST(LIT,|'|,|RA);
END; ELSE
DO; RT & "7F";
RT=RT>MAXOP THEN /* INVALID OPERATOR */
CALL ERROR(6); ELSE
DO CASE RT;

```



```

/* 0: UNASSIGNED */
CALL ERROR(0);
/* 1 ADD */
DO; RB=RB+RA; POP; STEP; TEST('ADD');
END;
/* 2 MIN */
DO; RB=RB-RA; POP; STEP; TEST('MIN');
END;
/* 3 MUL */
DO; RB=RB*RA; POP; STEP; TEST('MUL');
END;
/* 4 DIV */
DO; IF RA=0 THEN
  DO; CALL ERROR(10); RB=0;
  END; ELSE
  DO; I=RB; J=RA; I=I/J; RB=I; /* XPL RUNS OUT OF REGISTERS OTHERWISE */
  END;
  POP; STEP;
  TEST('DIV');
END;
/* 5 EXP */
DO; IF RA<0 THEN CALL ERROR(11); ELSE
  DO; IF RA=0 THEN RB=1; ELSE
    DO; I=RB;
      DO WHILE RA>1; RB=RB*I; RA=RA-1;
      END;
    END;
  END;
  POP; STEP; TEST('EXP');
END;
/* 6 NEG */
DO; RA=-RA; STEP; TEST('NEG');
END;
/* 7 RND */
DO; STEP;
  TEST('RND');
END;
/* 8 TRU */
DO; STEP;
  TEST('TRU');
END;
/* 9 LSS */
DO; IF RB<RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('LSS');
END;

```



```

/* LEQ */
DO; RB<=RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('LEQ');
END;
/* EQL */
DO; RB =RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('EQL');
END;
/* 12 NEQ */
DO; RB<=RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('NEQ');
END;
/* 13 GEQ */
DO; RB>=RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('GEQ');
END;
/* 14 GTR */
DO; RB> RA THEN RB=1; ELSE RB=0; POP; STEP; TEST('GTR');
END;
/* 15 NOT */
DO; IF RA=0 THEN RA=1; ELSE RA=0; STEP; TEST('NOT');
END;
/* 16 AND */
DO; I=RA; J=RB; IF (I=0)|(J=0) THEN RB=0; ELSE RB=1; POP; STEP;
TEST('AND');
END;
/* 17 BOR */
DO; IF RA=0 & RB=0 THEN RB=0 ; ELSE RB=1; POP; STEP; TEST('BOR');
END;
/* 18 IM1 */
DO; STEP; PUSH; RA=GETONE(RC);
INVALID;
STEP; TEST('IM1');
END;
/* 19 IM2 */
DO; STEP; PUSH; RA=GETTWO(RC);
INVALID;
STEP; STEP; TEST('IM2');
END;
/* 20 LOD */
DO; RA=M(RA); STEP; TEST('LOD');
END;
/* 21 STO */
DO; M(RB)=RA; RB=RA; POP; STEP; TEST('STO');
END;

```



```

/* PRO */
DO;
STEP; RG=RG-1; IF RG<=RS THEN CALL ERROR(12);
M(RG)=RC; RC=RA+B4; POP; INVALID; TEST('PRO');
END;
/* 23 RTN */
DO;
IF RG>=ST THEN CALL ERROR(13);
RC=M(RG); RG=RG+1; INVALID; /* ALREADY STEPPED */
TEST('RTN');
END;
/* 24 XIT */
DO; RC=0; TEST('XIT');
END;
/* 25 DEL */
DO; POP; STEP; TEST('DEL');
END;
/* DUP */
DO; PUSH; RA=RB; STEP; TEST('DUP');
END;
/* XCH */
DO;
I=RA; RA=RB; RB=I;
STEP; TEST('XCH');
END;
/* 28 INX */
DO; RA=RA+D; STEP; TEST('INX');
END;
/* 29 BFN */
DO; RC=RC+RA; POP; TEST('BFN');
INVALID;
END;
/* 30 BBN */
DO; RC=RC-RA; POP; TEST('BBN');
INVALID;
END;
/* 31 BFC */
DO; IF RB=0 THEN
DO; RC=RC+RA; INVALID;
END; ELSE STEP;
POP; POP; TEST('BFC');
END;
/* 32 BBC */
DO; IF RB=0 THEN
DO; RC=RC-RA; INVALID;
END; ELSE STEP;
POP; POP; TEST('BBC');
END;

```



```

/* 33 BRS */
DO; RC=RA+B4; POP; TEST('BRS');
INVALID;
END;
/* 34 BSC */
DO; IF RB=0 THEN
  DO; RC=B4+RA; INVALID;
  END; ELSE STEP;
  POP; POP; TEST('BSC');
END;
/* NOP */
DO; STEP; TEST('NOP');
END;
/* 36 GET */
DO;
RA=GETSPACE(RA); IF RAKO THEN CALL ERROR(14); STEP; TEST('GET');
END;
/* RET */
DO; CALL FORGETSPACE(RA); POP; STEP; TEST('RET');
END;
/* 38 RDV */
DO; PUSH; RA=READER; STEP; TEST('RDV');
END;
/* 39 WRV */
DO; CALL WRITE(0); POP; STEP; TEST('WRV');
END;
/* 40 WRS */
DO; CALL WRITE(1); POP; STEP; TEST('WRS');
END;
/* 41 DMP */
DO; IF LENGTH(OBUFF)>0 THEN CALL PRINT; STEP; TEST('DMP');
END;
/* 42 TAB */
DO; CALL TABULATE; POP; STEP; TEST('TAB');
END;
/* 43 SUB */
DO;
  CALCULATE A SUBSCRIPT */
  /* BASE OF ARRAY IS IN RA */
  L=RA; N=M(L); I=L; POP;
  DO J=1 TO N; POP;
    R=RA; I=I+R*M(L+J); /* LOOK AT LAST LINK WORD */
    END; J=F3&M(L-1); /* CHECK ADDRESS */
    RA=I; I=I+M(L); /* CHECK ADDRESS */
    STEP; IF (I>=J) | (I<=L) THEN CALL ERROR(16);
  END; TEST('SUB');
/* 44 BIF */

```



```

DO; J=RA;
IF (J<0) | (J>MAXINTR) THEN CALL ERROR(17); ELSE
DO CASE J;
TRACE=0;
DO; TRACE=1; CALL TRACEHEAD;
END;
CALL MEMDUMP;
/* SKIP (N) */
DO; POP; J=RA;
IF LENGTH(OBUFF)>0 THEN CALL PRINT;
DO WHILE J>0; J=J-1; OUTPUT='';
END;
END;
/* PAGE */
DO; IF LENGTH(OBUFF)>0 THEN CALL PRINT; OUTPUT(1)='1';
END;
/* APPEND */
DO; J=LENGTH(OBUFF);
IF J>0 THEN I=1; ELSE I=0;
DO WHILE I=1;
IF JK=0 THEN I=0; ELSE
IF BYTE(OBUFF,J-1)=-BYTE(' ') THEN I=0; ELSE J=J-1;
END;
RA=J+1;
END;
/* SET FIELDWIDTH */
DO; POP; J=RA;
IF (J<1) | (J>120) THEN CALL ERROR(19); ELSE
FIELDWIDTH=J;
END;
/* GET REGISTER S FOR THE "BEARD" */
RA=RS;
END;
/* RA IS FULL AT THIS POINT */ STEP; TEST('BIF');
END;
/* 45 ROW */
DO;
J=RB; /* J IS NUMBER OF SUBSCRIPTS */
IF RS-2*J-1 < SB THEN CALL ERROR(1);
I=RA; /* I IS NUMBER OF ARRAYS TO BE ALLOCATED WITH THESE BOUNDS */
POP; POP;
R=RS; N=1; /* SAVE RS--WE HAVE TO RESTORE LATER */
/* CALCULATE NUMBER OF FREE STORE CELLS REQ'D */
DO K=1 TO J;
U=RA; L=RB; POP; POP;
IF L>U THEN CALL ERROR(7);
N=N*(U-L+1);
END;

```


BNF FOR THE SAMPLE LANGUAGE

257

<TO>
 <TRANSFER>
 <ASSIGNMENT>
 <LEFT SIDE>
 <COMMENT>
 <BOOLEAN>
 <RELATION>

<EXPRESSION>

<TERM>

<FACTOR>

<PRIMARY>

<ARRAY VARIABLE>
 <INDEX>
 <TRAILER>

TO <EXPRESSION>
 JUMP <IDENTIFIER>
 <LEFT SIDE> <EXPRESSION>
 <PRIMARY> :=
 / * <REMARKS> * /
 <EXPRESSION> <RELATION> <EXPRESSION>
 LT LEQ NE GT
 <TERM>
 + - <TERM>
 <EXPRESSION> + <TERM>
 <EXPRESSION> - <TERM>
 <FACTOR>
 <TERM> * <FACTOR>
 <TERM> / <FACTOR>
 <PRIMARY>
 <FACTOR> ** <PRIMARY>
 <NUMBER>
 <IDENTIFIER>
 <ARRAY VARIABLE>
 <STRING>
 (<EXPRESSION>)
 <ARRAY NAME> <INDEX>
 <PRIMARY>
 END

APPENDIX J

CGP INPUT FOR THE SAMPLE LANGUAGE

```

$SYNTAX OUTPUT
$SYNTAX ITERATE
$SEMANTICS PUNCH
$SEMANTICS DUMP
BEGIN
/* SAMPLE LANGUAGE TEST OF CGP */
CELL A,B,ANAME;
FLAG DECL,CHARSTRING,NUMB;
TAG VECTR,INTR;
TABLE T1(250:TYPE BIT 8, LOCATION BIT 16);
<SAMPLE LANGUAGE> <HEADER> <BODY> <TRAILER>
CODE(OP 'XIT'); CODE(OP 'XIT'); CODE(OP 'XIT'); CODE(OP 'XIT');
<HEADER> <BEGIN> <DECLARATION SET>
DECL := FALSE; NUMB := TRUE;
<BEGIN> BEGIN
DECL := TRUE; CODE('1'); CODE(OP 'TRC'); CODE('2'); CODE(OP 'SUP');
<DECLARATION SET> <DECLARATION>;
* <DECLARATION SET> <DECLARATION>;
<DECLARATION>
<ARRAY>
<VARIABLE>
<ARRAY HEAD> <NUMBER> )
CODE('1'); CODE('FIXV(MPP1)'); CODE('1'); CODE('1'); CODE(OP 'ROW');
B := SET(A:LOCATION); CODE('8'); CODE(OP 'XCH'); CODE(OP 'STD');
<ARRAY HEAD>
ARRAY <ARRAY NAME>
<ARRAY NAME>
<IDENTIFIER> (
IF TEST DECL THEN DO; A := ENTRY(T1:=VAR(MP));
SET(A:TYPE;LOCATION) := (VECTR;GETPRT); END; ELSE DO;
A := LOC(T1:=VAR(MP)); ANAME := SET(A:LOCATION); END;
<VARIABLE>
VARIABLE <IDENTIFIER>
ENTER(T1:=VAR(SP);INTR;GETPRT);
<VARIABLE> <IDENTIFIER>
* ENTER(T1:=VAR(SP);INTR;GETPRT);
<BODY>
NOCODE;
* <BODY> <STATEMENT>

```



```

NOCODE;
<STATEMENT>
NOCODE;
*
NOCODE;
<LABEL>
FLOP(ADDR 'VAR(MP)') := CODELOC;
<SENTENCE>
NOCODE;
*
NOCODE;
<CONDITIONAL>
NOCODE;
*
NOCODE;
<IF THEN CLAUSE>
NOCODE;
*
CODELOC := FLOP('ELSE');
<IF THEN CLAUSE> <UNCONDITIONAL>
NOCODE;
*
CODELOC := FLOP('THEN');
<IF CLAUSE> <UNCONDITIONAL>
NOCODE;
*
FLOP('THEN') := CODELOC;
<IF ELSE CLAUSE> <UNCONDITIONAL>
NOCODE;
*
FLOP('ELSE') := CODELOC;
<IF CLAUSE> <UNCONDITIONAL>
NOCODE;
*
<DO GROUP> ;
NOCODE;
*
<TRANSFER> ;
NOCODE;
*
<ASSIGNMENT> ;
NOCODE;
*
<COMMENT>
NOCODE;
*
<INPUT>
NOCODE;
*
<OUTPUT>
NOCODE;
*
<INPUT>
READ <IDENTIFIER>
A := LOC(T1='VAR(SP)'); B := SET(A:LOCATION); CODE('B'); CODE(OP 'RDV');
CODE(OP 'STD');
*
CODE(OP 'RDV');
READ <ARRAY VARIABLE>
CODE(OP 'STD');
WRITE <PRIMARY>
<OUTPUT>
IF TEST CHARSTR 'WRS'); CODE(OP 'WRS'); CHARSTRING := FALSE; END;
ELSEDO; IF TEST NUMB THENDO; CODE(OP 'LOD'); END; CODE(OP 'WRV'); END;
CODE(OP 'DMP'); NUMB := TRUE;
<DO GROUP>
NOCODE;
*
<ITERATIVE DO>
NOCODE;

```



```

*      <TERM> * <FACTOR>
*      CODE(OP 'MUL');
*      <TERM> / <FACTOR>
*      <FACTOR>
*      IF TEST NUMB
*      <PRIMARY>
*      <FACTOR> THEN DO; CODE(OP 'LOD'); END; NUMB := TRUE;
*      <FACTOR> ** <PRIMARY>
*      IF TEST NUMB THEN DO; CODE(OP 'LOD'); END; CODE(OP 'EXP'); NUMB := TRUE;
*      <PRIMARY>
*      CODE('FIXV(MP)'); NUMB := FALSE;
*      <IDENTIFIER>
*      A := LOC(T1='VAR(MP)'); B := SET(A:LOCATION); CODE('B');
*      NOCODE;
*      <STRING>
*      CODE 'A=STORESTRING(VAR(MP));'; CODE('A'); CODE(OP 'INX');
*      CHARSTRING := TRUE;
*      ( <EXPRESSION> )
*      NOCODE;
*      <ARRAY VARIABLE> <ARRAY NAME> <INDEX>
*      CODE('ANAME'); CODE(OP 'LOD'); CODE(OP 'SUB');
*      <INDEX> <PRIMARY>
*      IF TEST NUMB THEN DO; CODE(OP 'LOD'); END; NUMB := TRUE;
*      <TRAILER>
*      NOCODE;
*      END
EOF

```


CGP OUTPUT FOR THE SAMPLE LANGUAGE

一、

263

264


```

CASE 46 **// DO; CALL EMITX(FIXV(MPPL) | "80"); END;
CASE 47 **// DO; FIXV(MP)=LSS; END;
CASE 48 **// DO; FIXV(MP)=LEQ; END;
CASE 49 **// DO; FIXV(MP)=LEQ; END;
CASE 50 **// DO; FIXV(MP)=NEQ; END;
CASE 51 **// DO; FIXV(MP)=GEQ; END;
CASE 52 **// DO; FIXV(MP)=GTR; END;
CASE 53 **// DO; END;
CASE 54 **// DO; END;
CASE 55 **// DO; CALL EMITX(NEG | "80"); END;
CASE 56 **// DO; CALL EMITX(ADD | "80"); END;
CASE 57 **// DO; CALL EMITX(MIN | "80"); END;
CASE 58 **// DO; END;
CASE 59 **// DO; CALL EMITX(MUL | "80"); END;
CASE 60 **// DO; CALL EMITX(DIV | "80"); END;
CASE 61 **// DO; IF NUMB THEN DO; CALL EMITX(LOD | "80"); END; NUMB=TRUE; END;
CASE 62 **// DO; IF NUMB THEN DO; CALL EMITX(LOD | "80"); END;
CASE 63 **// DO; CALL EMITX(EXP | "80"); NUMB=TRUE; END;
CASE 64 **// DO; CALL EMITX(FIXV(MP)); NUMB=FALSE; END;
CASE 65 **// DO; A=LOOKUP(VAR(MP), T1); A=POINTR(A); B=LOCATION(A); CALL EMIT(B);
CASE 66 **// DO; END;
CASE 67 **// DO; A=STORESTRING(VAR(MP)); CALL EMIT(A); CALL EMITX(INX | "80");
CASE 68 **// DO; END;
CASE 69 **// DO; CALL EMITX(SUB | "80"); END;
CASE 70 **// DO; IF NUMB THEN DO; CALL EMITX(LOD | "80"); END; NUMB=TRUE; END;
CASE 71 **// DO; END;

```


LIST OF REFERENCES

1. O'Neil, John T. Jr., Meta-Pi: An On-Line Interactive Compiler-Compiler, p. 201-213, Proceedings AFIPS FJCC 1968.
2. Feldman, Jerome A., et al., Translator Writing Systems, Technical Report No. CS69, Computer Science Department, Stanford University, 9 June 1967.
3. Rosen, Saul, Programming Systems and Languages, p. 265-358, McGraw-Hill, 1967.
4. Feldman, Jerome A., A Formal Semantics for Computer Oriented Languages, Ph.D. Thesis, Carnegie Institute of Technology, 1964.
5. McKeeman, William M., et al., A Compiler Generator, Prentice-Hall, Inc., 1970.
6. Wirth, N., and Weber, H., EULER: A Generalization of Algol, and its Formal Definition: Part I, CACM, v. 9, p. 13-25, January 1966.
7. Kildall, Gary A., ALGOL-E, Technical Report No. NPS-53KD71051A, Department of Mathematics, Naval Postgraduate School, Monterey, June 1971.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3. LT Gary A. Kildall, Code 53KD (thesis advisor) Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
4. LT Peter C. Finne, USN (student) 1628 South Seventh Street Brainerd, Minnesota 56401	1

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Naval Postgraduate School Monterey, California 93940		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE XPL CGP: An XPL-based Semantic Language Processor			
4. DESCRIPTIVE NOTES (Type of report and, inclusive dates) Master's Thesis; June 1971			
5. AUTHOR(S) (First name, middle initial, last name) Peter Charles Finne			
6. REPORT DATE June 1971		7a. TOTAL NO. OF PAGES 272	7b. NO. OF REFS 7
8a. CONTRACT OR GRANT NO.		9a. ORIGINATOR'S REPORT NUMBER(S)	
b. PROJECT NO.			
c.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.			
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Naval Postgraduate School Monterey, California 93940	
13. ABSTRACT <p>The XPL CGP is a complete compiler generator package based on the XPL system. With the introduction of a semantic meta-language (SML) and an associated processor, the package is capable of generating a production compiler for any computer language with a mixed strategy precedence grammar. The only input required in most cases is the syntax of the language encoded in BNF and the corresponding semantics encoded in SML. The resulting compiler will generate code which may be executed on a simulated stack-oriented machine.</p>			

Semantic Metalanguage

LINK A

LINK B

LINK C

ROLE

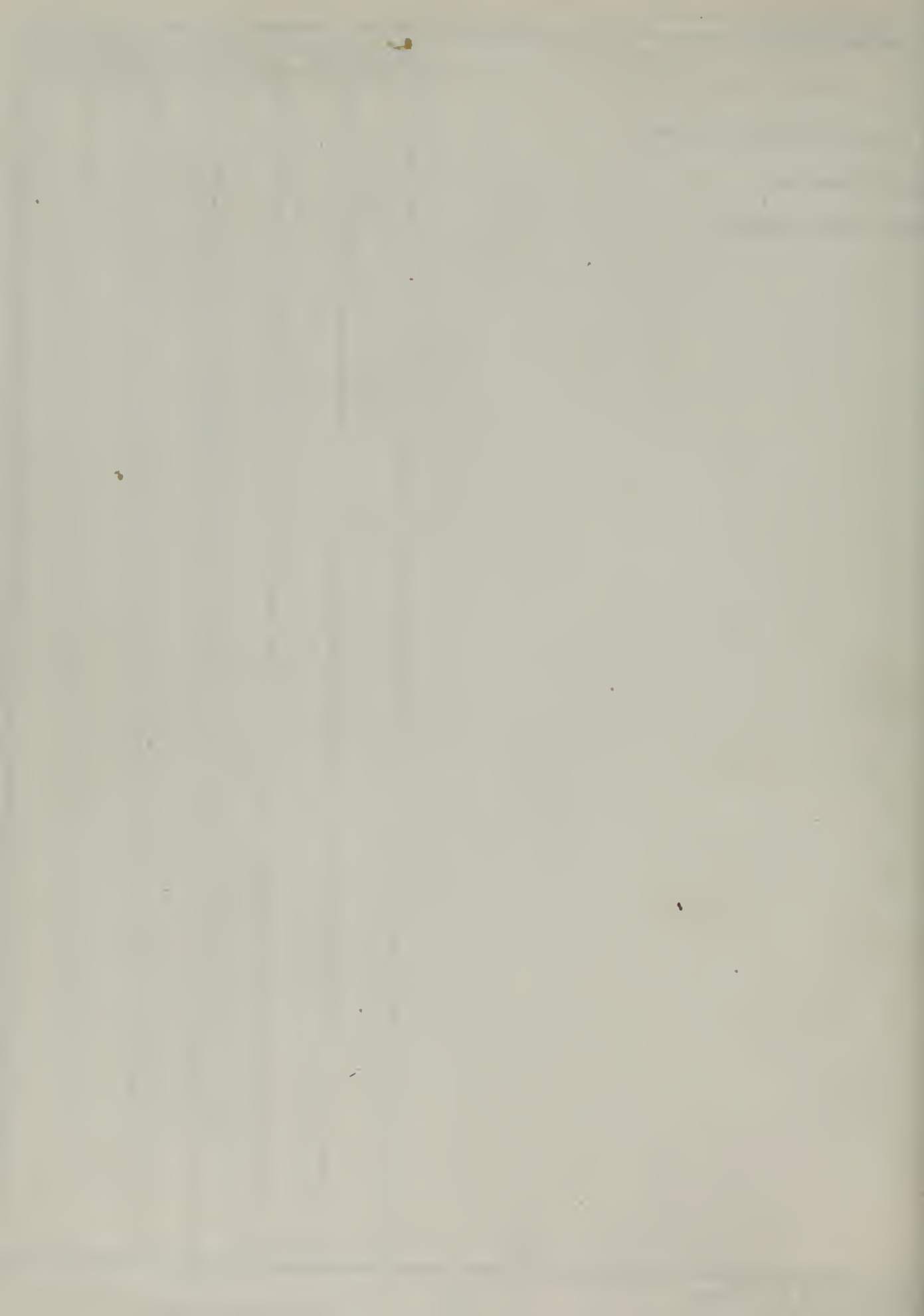
WT

[illegible]

WT

HOLE

WT



BINDERY

Thesis
F4454 Finne
c.1

128587

XPL CGP: An XPL-based
semantic language pro-
cessor.

BINDERY

Thes
F445
c.1

Thesis
F4454 Finne
c.1

128587

XPL CGP: An XPL-based
semantic language pro-
cessor.

thesF4454

XPL CGP :



3 2768 002 00168 7

DUDLEY KNOX LIBRARY